Generative Software-Entwicklung zur numerischen Strömungssimulation

Von der Fakultät für Mathematik Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

genehmigte Dissertation

vorgelegt von

Dipl.-Math. ***Volodymyr Myrnyy***

geboren am 17.07.1978 in Dnipropetrowsk, Ukraine

Gutachter: Prof. Dr. Michael Fröhner

Gutachter: Prof. Dr. Peter Bachmann

Gutachter: Prof. Dr. Arnd Meyer

Gutachter: Prof. Dr. Michael Griebel

Tag der mündlichen Prüfung: 21. Februar 2006

Für meine Eltern

Danksagung

Ich bedanke mich vor allem bei meinem Betreuer Prof. Dr. Fröhner für die Überlassung der Aufgabenstellung und den großen Freiraum bei der Durchführung der Arbeit. Diskussionen, Kritik und Hilfsbereitschaft unterstützten die Ansiedlung der Programmierarbeiten auf dem Gebiet der numerischen Strömungsmechanik. Besonders dankbar bin ich für die Vermittlung wichtiger und hilfreicher Beziehungen innerhalb und außerhalb der Universität sowie nützlicher internationaler Kontakte.

Mein herzlicher Dank gilt Frau Prof. Dr. Pickenhain und Frau Prof. Dr. Bordag, die ein BMBF-Projekt zur Strömungssimulation leiteten. Im Rahmen meiner Arbeit an diesem Projekt und in Zusammenarbeit mit den russischen Wissenschaftlern Prof. Chkhetiani und Prof. Ponomarev (Space Research Institut, Moscow), Prof. Shokin und Prof. Chubarov (Institut of Computational Technologies, Novosibirsk) und anderen Wissenschaftlern konnte ich zahlreiche praktische Erkenntnisse und Erfahrungen sammeln. Ich bin den Kollegen aus dem Rechenzentrum Stuttgart (HLRS) und der Technischen Universität Chemnitz dankbar, die mir die Möglichkeit gaben, an modernen Parallelrechnern, Supercomputern und PC-Clustern zu arbeiten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einführung

Der Anfang ist die Hälfte des Ganzen.
- Aristoteles

Die Bedeutung der numerischen Simulation bei der Bearbeitung von Problemen echter praktischer Relevanz kann man zur Zeit nicht hoch genug einschätzen, da mehr und mehr teure und langwierige Experimente und Versuche durch Computersimulationen ersetzt werden können. Dabei tauchen neue und anspruchsvolle Aufgaben parallel zur sich schnell entwickelnden Rechentechnik und zum steigenden Forschungsbedarf auf, so dass noch mehr Wissenschaftler in dieses Forschungsgebiet einbezogen werden, mit einer Problemvielfalt, die noch längst nicht ausgeschöpft ist.

Unter diesen Problemen spielt die Strömungssimulation eine besondere Rolle, weil das zugehörige mathematische Modell, d.h. die Navier-Stokes-Gleichungen mit zugehörigen Rand- und Anfangsbedingungen, noch viele offene Fragen bei der analytischen und numerischen Behandlung aufweist sowie eine große Rechenleistung zur Simulation reeller dreidimensionaler Strömungsvorgänge erfordert.

1.1 Motivation und Ziele

Man gelangt von den Navier-Stokes-Gleichungen oder deren Sonderfällen und ihrer analytischen Untersuchung über die große numerische Verfahrensvielfalt der Zeit-, Raumdiskretisierung und Gleichungssystemlösung zur Softwareentwicklung, die die Simulation verwirklicht. Auf diesem Weg müssen die besten bekannten Ergebnisse und Erfahrungen auf jedem Schritt ausgewählt werden, da ein Engpass an einer Stelle die gesamte Anwendung zum Misserfolg führen kann. Obwohl die größten Effizienzfortschritte in der richtigen Wahl und Weiterentwicklung moderner numerischer Verfahren liegen, müssen auch aktuelle Programmiertechniken studiert und angewendet werden. Diese Voraussetzungen begründen den Schwerpunkt dieser Arbeit: Auswahl, Untersuchung und Vergleich numerischer Verfahren und effiziente Softwareentwicklung zur Simulation inkompressibler Newtonscher Strömungen in ein-, zwei- und dreidimensionalen Gebieten, die mit unstrukturierten Gittern diskretisiert wurden.

Das instationäre Navier-Stokes-Problem wird bezüglich der Zeit implizit behandelt, ohne die Geschwindigkeits-Druck-Kopplung zu zerlegen (Fractional-Step-Methoden, Abschnitt 3.1.1). Für die Raumdiskretisierung werden die Galerkin-

sche Finite-Elemente-Methode (FEM) bevorzugt, nachdem die Äquivalenz zwischen Finite-Volumen-Methode (FVM) und einigen stabilen Stokes-Elementen im Abschnitt (3.5) bewiesen wurde.

1.2 Generative Softwareentwicklung und Wissenschaftliches Rechnen

Wenn ein Softwaresystem in einem bestimmten mathematischen Gebiet entwickelt wird, möchte man eine gewisse Flexibilität erreichen, um verschiedene Verfahren durch Parametervariation behandeln zu können und dadurch Experimente und Untersuchungen zu ermöglichen. Außerdem muss die Software für eine Weiterentwicklung offen sein, ohne komplizierte Veränderungen in den vorhandenen Softwarekomponenten vornehmen zu müssen.

In ***objektorientierten*** Programmiersprachen werden diese Ziele durch die Klassenabstraktion erreicht, wobei jede Klasse einen Datensatz mit dazugehörigem Bearbeitungsverfahren als einen neu definierten Datentyp darstellt. Das gesamte Problem wird dann von oben nach unten (Top-Down-Technik) bis in die einzelnen Komponenten hierarchisch zerlegt, die als Klassen implementiert werden. Die endgültige Definition von Komponenten wird durch die Problemstellung verursacht, weil man sich bei der Entwicklung nur mit der gestellten Aufgabe befasst und an eine mögliche Weiterentwicklung oder einen Einsatz der Software für andere Aufgaben nicht denkt. Das kann dazu führen, dass trotz des objektorientierten Designs die Komponenten nur schlecht oder gar nicht wiederverwendbar sind.

Eine wichtige Ergänzung zur objektorientierten Programmierung (OOP) ist die sog. ***generische Programmierung***. Eine Klasse kann in der generischen Programmierung einen oder mehrere Parameter bekommen und heißt danach Klassen-Muster oder ***Klassen-Template***. Ein Klassen-Template stellt damit eine Menge von Klassen dar. In der Kompilierungszeit werden aus Klassen-Templates durch die Eingabe von Template-Parameter gewöhnliche Klassen erzeugt (Abschnitt 4.1.1). Ein oder mehrere Parameter können auch Funktionen zugeordnet werden, um mit einer gleichen Definition Argumente unterschiedlicher Typen akzeptieren zu können. Diese Fähigkeit gab den Ursprung zum algorithmuszentrierten Ansatz in der wissenschaftlichen Software. Dabei wird der Fokus von Klassen auf Algorithmen übertragen, die insbesondere im wissenschaftlichen Rechnen mehr Bedeutung als die Datenspeicherung haben können. Eine Funktion kann z.B. den gleichen Algorithmus mit unterschiedlichen Daten ausführen, ohne überhaupt die Information zu besitzen, wie die Daten gespeichert sind und was sie repräsentieren. Der Datenzugriff erfolgt dabei über sog. ***Iteratoren***, die mit einem gleichen Interface für die zu bearbeitenden Klassen implementiert sind. Der Hauptvertreter dieser Idee in der Programmiersprache C++ ist die Standard Template Library (STL) von Alex Stepanov. In der Arbeit [18, 19] wird diese Idee zur Behandlung von Gitter-Datenstrukturen verwendet. Es gibt aber auch einen Nachteil, wenn die sog. Abstraktionsstrafe (engl. abstaction penalty) groß werden kann, weil das gleiche Iteratoren-Interface nicht für alle Klassen optimal implementiert werden kann.

Die ***generative Programmierung*** (GP) (eingeführt in den Arbeiten von Czarnecki und Eisenecker [37, 38, 39, 43]) nutzt auch Klassen-Templates und schließt damit die generische Programmierung mit ein. Darüber hinaus werden im Rahmen

der GP besondere Klassen-Templates umgesetzt, die ausschließlich eigene Template-Parameter behandeln und deshalb nur in der Kompilierungszeit eine Wirkung haben. Solche Klassen-Templates werden Generatoren genannt, weil hinter den Template-Parametern auch Klassen bzw. Klassen-Templates stehen können, die in sich Daten und Algorithmen tragen. Dieser Prozess heißt auch ***statische Konfiguration*** des Programmcodes.

Wir betrachten nun ein einfaches Beispiel, um die Unterschiede im Code-Aufbau durch statische Konfiguration und bei der normalen OOP zu veranschaulichen (Tab. 1.1). Es soll ein Programm zur Lösung der Navier-Stokes- und Stokes-Gleichungen mit gemischten finiten Elementen P_2/P_0 und P_2/P_1 entwickelt werden. Das Gitter wird durch eine Datenstruktur M repräsentiert und das lineare Gleichungssystem mit GMRES-Verfahren gelöst. Aus Gründen der Einfachheit vernachlässigen wir Präkonditionierer und weitere Einzelheiten, die ein solches Programm enthalten muss. Jeder der genannten Programmteile wird als eine Klasse bzw. Klassenhierarchie gemäß der OOP implementiert und in der Tabelle 1.1 als ein Kästchen mit entsprechenden Buchstaben dargestellt. Im objektorientierten Programm (Tab. 1.1, 2. Spalte) wird ***dynamische Bindung*** zwischen diesen Programmteilen entwickelt, damit eine der beiden Gleichungen gewählt wird und mit jeweils einem oder anderem finiten Element gelöst werden kann. In einer geschickten Implementierung sind die Rechenkosten der dynamischen Bindung gering, sie existieren aber bei jedem Aufruf dieses Programms. In der generativen Version (Tab. 1.1, 3. Spalte) werden fertige Solver (Spalten im Bild) aus den vorhandenen Bauteilen vom Compiler konstruiert, wobei ein, mehrere oder alle möglichen Solver in der Kompilierungszeit erstellt bzw. konfiguriert werden können. Ein ausgewählter Solver läuft also ganz ohne überflüssige dynamische Bindung ab.

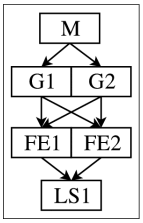
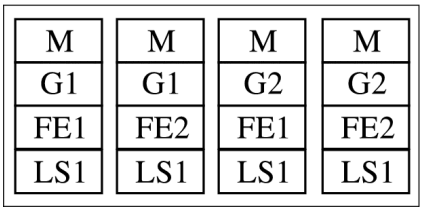
	Bauteile	OOP	GP
M	Gitter		
G1	NaSt-Gleichungen		
G2	Stokes-Gleichungen		
FE1	FEM P_2/P_0		
FE2	FEM P_2/P_1		
LS1	GMRES		

Tabelle 1.1: Code-Aufbau einer objektorientierten und einer generativen Anwendung

Was passiert mit dem Beispiel-Programm, wenn noch weitere Methoden eingefügt werden müssen, was evtl. für einen großen Simulator gewünscht sein kann? Die Methoden werden als weitere Programm-Bauteile implementiert und müssen mit den anderen Teilen kompatibel sein. In der OOP-Version muss dann die dynamische Bindung geändert werden, deren Umfang mit jeder neuen Methode exponenziell steigt. Noch mehr dynamische Bindung liefern unvermeidbare Bedingungen an den Methoden-Einsatz. Beispielsweise muss das GMRES-Verfahren für die Navier-Stokes-Gleichungen benutzt werden, wobei für die Stokes-Gleichungen das MINRES-Verfahren viel günstiger ist (Abschnitt 5.2.5). In der GP-Version steigt dagegen die Anzahl der möglichen Solver, die genau so effizient bei der steigenden Methoden-Anzahl bleiben. Die Bedingungen an den Methoden-Einsatz werden schon

in der Kompilierungszeit (statisch) erfüllt, d.h. unzulässige Solver werden nicht erstellt. Damit die Bauteile noch besser kombiniert werden können, zerlegt man sie in elementare Komponenten. So haben z.B. Stokes- und Navier-Stokes-Gleichungen viel gemeinsam und werden in die Gleichungsterme zerlegt (Abschnitt 4.2). Dann unterscheiden sich die Gleichungsmodelle nur bis auf den konvektiven Term.

Eine solche Zerlegung eines Problems in die Komponenten wird in der ***Domain-Engineering-Theorie*** betrachtet, die ein Bestandteil der GP ist. Der Blick wird breiter von einem Softwaresystem zur Lösung einer bestimmten Aufgabe auf die gesamte Softwaresystemfamilie geworfen. Die über die gesamte Softwaresystemfamilie veränderlichen Eigenschaften helfen bei der richtigen Bestimmung von Parametern, die zu den elementaren wiederverwendbaren Komponenten führen. Solche Komponenten können dann mit Hilfe von Generatoren in fertige Softwaresysteme (hier auch Solver) dieser Softwaresystemfamilie automatisch zusammengestellt werden. Zusammengefasst kann der Entwicklungsprozess grob in zwei Bestandteile zerlegt werden [190]:

- Entwicklung für die Wiederverwendung: Zunächst werden Infrastruktur und Komponenten entwickelt. Es wird die Basis bzw. die Generatoren für die Entwicklung von Systemvarianten realisiert. Dieser Bestandteil des Entwicklungsprozesses wird als ***Domain-Engineering*** bezeichnet.
- Entwicklung mit Wiederverwendung: Anschließend werden die vorhandenen Komponenten und die Infrastruktur zu einer neuen Systemvariante zusammengefügt, es werden also die eigentlichen Anwendungen zusammengestellt. Dieser Bestandteil wird deshalb als ***Application-Engineering*** bezeichnet.

Die Softwareentwicklung nach diesen Richtlinien erhält sowohl eine hohe Flexibilität als auch Effizienz, weil die Generatoren nur in der Kompilierungszeit laufen und keinen Aufwand für die Laufzeit-Programme bedeuten. Diese Eigenschaften entsprechen sehr gut den Anforderungen der wissenschaftlichen Software.

Diese Arbeit beschäftigt sich mit der Softwareentwicklung im Problembereich (in der ***Domäne***) 'Strömungssimulations-Solver', der zwischen einem vorhandenen Gittergenerator und Postprozessor (Darstellung von Resultaten) liegt, und setzt sich die folgenden Ziele:

1. Generatives Design: Ein hochgradig angepasster und optimierter Strömungssimulator kann ausgehend von einer Anforderungsspezifikation mittels Konfigurationswissen aus elementaren wiederverwendbaren Komponenten nach Bedarf automatisch erzeugt werden. Voneinander unabhängige Komponenten (Klassen) bilden damit unterschiedliche Ansätze zur Lösung der Navier-Stokes-Gleichungen und bieten dadurch eine breite Palette numerischer Methoden.
2. Höchste Effizienz bzgl. Rechenzeit und Speicherverbrauch, um große Strömungsprobleme (über 10^5 Gitter-Elemente) auch auf einem Prozessor in endlicher Zeit simulieren zu können.
3. Plattformunabhängige Implementierung: das Softwaresystem muss auf unterschiedlichen Hardware-Plattformen so funktionieren, dass auch der vorhergehende Punkt möglichst erfüllt wird.

Die höchste Programmleistung kann nach den folgenden Regeln erreicht werden:

1. Auswahl optimaler Algorithmen ist die wirksamste Methode. Zum Beispiel kann eine Umstellung von Gliedern oder eine Zerlegung in Faktoren die Berechnung einer Matrix-Vektor-Beziehung wesentlich vereinfachen (z.B. Abschnitt 5.2.4). Viel komplizierter ist beispielsweise eine Auswahl von Solver und Präkonditionierer linearer Gleichungssysteme, deren optimale Parameter meist nur heuristisch durch Tests eingestellt werden können.
2. Übergang von komplizierten Datenstrukturen (z.B. eines Gitters) zu Arrays bzw. zu Vektoren und Matrizen, weil moderne Prozessoren entsprechend ihrer Bauweise mehr als eine Operation pro Takt ausführen können. Das funktioniert aber normalerweise nur mit Arrays [93] (siehe auch Tests im Abschnitt 5.3.3).
3. Vermeiden von mehrfacher Berechnung gleicher Formeln in Schleifen. Das gilt sowohl für kleine Algorithmen als auch für globale Konzepte, wie die Steifigkeitsmatrix-Assemblierung (Abschnitt 5.2.4).
4. Eine korrekte Speicher-Verwaltung. Speicher-Operationen sind aufwendig, deshalb muss die Speicher-Zuteilung minimiert oder aus Schleifen vollständig ausgeschlossen werden. Für diesen Zweck muss der maximale Speicher-Bedarf verschiedener Datenstrukturen vor der Anwendung in einer Schleife abgeschätzt werden. Ein gutes Beispiel hierfür ist die Größe der schwachbesetzten Steifigkeitsmatrix (Abschnitt 5.2.4), die vor der Assemblierung anhand der Gitterdaten exakt bestimmt werden kann.
5. Programmiersprache- und Compiler-abhängige Optimierungen des Programm-Codes. Ein Beispiel ist das manuelle Loop-Unrolling mittels Template-Metaprogrammierung (Abschnitt 5.3.3).

Zur Untersuchung und zum Testen der Rechenleistung kann die vorhergehende Liste in die folgenden zwei Aspekte umgeordnet werden:

- Minimierung der Anzahl der mathematischen Operationen bzw. der Algorithmen-Komplexität (Regeln 1 und 3).
- Maximierung der Prozessorleistung (Regeln 2,4 und 5) bzw. Erreichen eines größeren Anteils der theoretischen Höchstleistung (engl. peak performance). Diese Größe wird in MFLOPS (engl. millions of floating-point operations per second) oder in normalisierten MFLOPS gemessen (Anhang A.3).

1.3 Aufbau der Arbeit

Kapitel 2

Die allgemeine Form der Navier-Stokes-Gleichungen, bestehend aus der Impulsgleichung, der Kontinuitätsgleichung und der Energiegleichung, wird in diesem Kapitel vorgestellt. In dieser Form werden die Fluideigenschaften mit Hilfe der Kennzahlen diskutiert (Abschnitt 2.1). Danach wird das Modell auf die isoenergetische und inkompressible Formulierung reduziert, die im weiteren Verlauf der Arbeit benutzt wird. Darüber hinaus werden die Sonderfälle, wie axialsymmetrische Strömung, Stokes- und Euler-Gleichungen usw. zusammengefasst (Abschnitt 2.2), wobei nur

Formulierungen in den primitiven Variablen ($\boldsymbol{u}$-p-Formulierungen) mit unterschiedlichen Randbedingungen (Abschnitt 2.3) berücksichtigt werden.

Kapitel 3

Die $\boldsymbol{u} - p$-Formulierung des Navier-Stokes-Problems wird zuerst im Abschnitt (3.1) bzgl. der Zeit diskretisiert und linearisiert. Für die nachfolgende Umsetzung wird das Einschritt-θ-Schema bevorzugt. Im Unterschied zu den Projektionsmethoden [71, 72, 137, 160], die den Druck von der Geschwindigkeit separieren, approximieren die θ-Schemata den Druck wesentlich besser, obwohl die Dimension des dabei entstehenden Gleichungssystems größer ist.

Im Abschnitt (3.2) werden die wichtigsten Eigenschaften einer Delaunay-Triangulierung in zwei und drei Dimensionen sowie des dualen Voronoi-Diagramm beschrieben. Nach einer kurzen Einführung in die Finite-Volumen- (Abschnitt 3.3) und die Finite-Elemente-Methoden (Abschnitt 3.4) werden diese Eigenschaften bei einer FVM-FEM-Äquivalenz-Untersuchung im Abschnitt (3.5) verwendet. Im Vergleich zu den anderen FVM/FEM-Übergängen für elliptische Probleme [78, 128, 167, 168] oder für die Petrov-Galerkin-FEM [187, 98] wird die Untersuchung auf die LBB-stabilen Galerkin-FEM beschränkt. Dies limitiert die FEM-Vielfalt, wobei die Ansatzfunktionen für die Geschwindigkeit mindestens zweiter Ordnung oder auf verfeinertem Gitter definiert werden müssen. Die Betrachtung wird auf Delaunay-Triangulierungen in zwei Dimensionen ausführlich durchgeführt. Neben der bekannten Konservativitätsaussage für die Stokes-Elemente mit unstetigem Druck (P_2/P_0, P_1^M/P_0) werden die äquivalenten FVM formuliert, die die Konvergenz- und Stabilitätseigenschaften der theoretisch gut unterstützten GFEM übernehmen.

Kapitel 4

Nach einer kurzen Einführung in die GP wird die Template-Metaprogrammierung in der Sprache C++ beschrieben, die die bedeutensten Implementierungstechniken der GP darstellt. Eine solche Entwicklung von Modellen und Programmen, die nur in der Kompilierungszeit existieren, findet eine vorteilhafte Anwendung für mathematische Objekte und Verfahren [122, 123], wenn alle notwendige Daten schon vor der Kompilierung vorhanden sind. So können Navier-Stokes-Gleichungen symbolisch modelliert und behandelt werden sowie unterschiedlichen Zeitdiskretisierungs- und Linearisierungs-Verfahren auf dieses Gleichungs-Modell angewendet werden (Abschnitt 4.2). Die polynomialen Ansatzfunktionen einer bestimmten Ordnung auf einem bestimmten Geometrie-Referenzelement sind ebenso vor der Kompilierung bekannt. Deshalb können sie und notwendige Integrale auch durch Metaprogramme berechnet werden (Abschnitt 4.3). Es wird auch eine weniger komplizierte Implementierung der statischen Konfiguration am Beispiel gemischter Stokes-Elementen dargestellt (Abschnitt 4.4).

Kapitel 5

Dieses Kapitel beschäftigt sich mit den theoretischen Software-Entwicklungsmethoden der GP, indem der Problembereich 'Strömungssimulations-Solver' im Rahmen der Domain-Engineering in drei Etappen beschrieben wird: Analyse, Design und

Implementierung. Die Modellierung wird mit Hilfe von Merkmaldiagrammen dargestellt, die die Hierarchie von Konzepten und Merkmalen grafisch erklären. Die Hauptkonzepte (Gitter, diskrete Gleichungen, FEM, Assemblierung, Gleichungssystemlösung und Präkonditionierung, Parallelisierung, Fehlerschätzung) werden zusammen mit aktuell vorhandenen numerischen Verfahren entwickelt und bis auf elementare Merkmale zerlegt (Abschnitt 5.2).

Ein besonderer Schwerpunkt liegt in der Einteilung sämtlicher Daten in die ***statischen*** (diese können in der statischen Konfiguration teilnehmen) und die ***dynamischen***, die erst in der Laufzeit zur Verfügung stehen (Abschnitt 5.1).

Der Abschnitt (5.3) stellt besonders interessante Teile der Implementierung und ausgewählte Performance-Tests zusammen, wobei die Implementierungen mit Hilfe der Template-Metaprogrammierung aus dem vierten Kapitel eine logische Fortsetzung dieses Abschnitts sind.

Kapitel 6

Einige numerische Strömungssimulationen werden in diesem Kapitel so durchgeführt, dass mehrere Fähigkeiten des entwickelten Simulationssolvers getestet und vorgestellt werden können.

Kapitel 7

Die wichtigsten Schwerpunkte, Ergebnisse und Schlussfolgerungen der durchgeführten Arbeit werden zusammengefasst sowie einige mögliche Richtungen der Weiterentwicklung genannt.

Kapitel 2

Navier-Stokes-Gleichungen

Gleichungen sind wichtiger für mich, weil die Politik für die Gegenwart ist, aber eine Gleichung etwas für die Ewigkeit.
- Albert Einstein

Es werden die Bewegungsgleichungen eines Fluids in diesem Kapitel zusammengefasst. Sei bei einer dreidimensionalen Bewegung das Strömungsfeld durch den Geschwindigkeitsvektor $\boldsymbol{u} = (u, v, w)^T$, den Druck p und die Temperatur T bestimmt, falls sich diese Größen in der Zeit verändern. Zur Bestimmung dieser fünf Größen stehen folgende Gleichungen zur Verfügung [67, 95, 150]:

- Kontinuitätsgleichung - Erhaltung der Masse:

$$\frac{D\rho}{Dt} + \rho \nabla \cdot \boldsymbol{u} = 0, \tag{2.1}$$

$\frac{D\rho}{Dt} = \frac{\partial \rho}{\partial t} + \boldsymbol{u} \cdot \nabla \rho$ - totale oder substanzielle Ableitung der Dichte ρ nach der Zeit

- Impulsgleichungen - Erhaltung des Impulses oder Navier-Stokes-Gleichungen im engeren Sinne:

$$\rho \frac{D\boldsymbol{u}}{Dt} = \boldsymbol{f} - \nabla p + \nabla \cdot \tau, \tag{2.2}$$

$\frac{D\boldsymbol{u}}{Dt} = \frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u}$ - ***substanzielle*** oder ***totale Beschleunigung***,

τ - ***viskoser Spannungstensor***.

- Energiegleichung: Sie drückt die Erhaltung der Energie aus:

$$\rho c_p \frac{DT}{Dt} = \nabla \cdot (\sigma \nabla T) + \beta T \frac{Dp}{Dt} + \Phi, \tag{2.3}$$

σ - ***Wärmekapazität*** (eine positive Stoffgröße),

c_p - ***isobare spezifische Wärmekapazität***,

$\beta = -\frac{1}{\rho}\left(\frac{\partial \rho}{\partial T}\right)_p$ - ***Wärmeausdehnungskoeffizient***,

$\Phi = \nabla \cdot (\tau \boldsymbol{u}) - \boldsymbol{u} \nabla \cdot \tau$ - ***Dissipationsfunktion***.

Die Gleichungen (2.1)-(2.3) gelten in einem beliebigen Koordinatensystem. Der viskose Spannungstensor $\tau = \tau_{ij}$ $(i, j \in \{x, y, z\})$ wird im dreidimensionalen kartesischen Koordinatensystem durch folgende Matrix repräsentiert:

$$\begin{aligned}
\tau_{xy} = \tau_{yx} &= \eta\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right), & \tau_{xx} &= \lambda\nabla\cdot\boldsymbol{u} + 2\eta\frac{\partial u}{\partial x},\\
\tau_{yz} = \tau_{zy} &= \eta\left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z}\right), & \tau_{yy} &= \lambda\nabla\cdot\boldsymbol{u} + 2\eta\frac{\partial v}{\partial y},\\
\tau_{zx} = \tau_{xz} &= \eta\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right), & \tau_{zz} &= \lambda\nabla\cdot\boldsymbol{u} + 2\eta\frac{\partial w}{\partial z},
\end{aligned} \tag{2.4}$$

Die Proportionalitätsfaktoren η und λ heißen ***Viskosität*** und ***Massenviskosität*** und werden nach der Stokes-Hypothese $\lambda = -\frac{2}{3}\eta$ zusammengesetzt.

2.1 Kennzahlen

Wir gehen von den Bewegungsgleichungen aus und führen dimensionslose Größen für jede Variable ein, die in den Bewegungsgleichungen auftreten. Als Referenzgrößen wählen wir eine Bezugslänge l (typische Körperabmessung), eine Bezugsgeschwindigkeit V (z.B. die Anströmgeschwindigkeit) und einen thermodynamischen Bezugszustand, gekennzeichnet durch die Referenz-Temperatur T^* und den Referenz-Druck p^*. Die dazugehörigen Stoffwerte sind $\rho^*, \eta^*, c_p^*, \lambda^*$ und β^*. Für die Massenkraft pro Volumeneinheit wird gesetzt

$$\boldsymbol{f} = \rho g \boldsymbol{e}_g,$$

wobei $\boldsymbol{e}_g$ der Einheitsvektor in Richtung der Fallbeschleunigung ist und g als Konstante angenommen wird. Werden die dimensionslosen Größen in die Bewegungsgleichungen eingesetzt, so ergibt sich:

$$\left\{\begin{aligned}
\frac{D\rho^*}{Dt^*} &= -\rho^*\nabla\cdot\boldsymbol{u}^*,\\
\rho^*\frac{D\boldsymbol{u}^*}{Dt^*} &= \tfrac{1}{\mathrm{Fr}^2}\rho^*\boldsymbol{e}_g - \nabla p^* + \tfrac{1}{\mathrm{Re}}\nabla\cdot\tau^*,\\
\rho^* c_p^*\frac{DT^*}{Dt^*} &= \tfrac{1}{\mathrm{RePr}}\nabla\cdot(\lambda^*\nabla^* T^*) - \mathrm{K}_\rho\mathrm{Ec}\beta^* T^*\tfrac{Dp^*}{Dt^*} + \frac{\mathrm{Ec}}{\mathrm{Re}}\Phi^*,
\end{aligned}\right. \tag{2.5}$$

wobei fünf dimensionslose ***Kennzahlen***, die in Tabelle 2.1 erläutert werden, auftreten. Zwei Strömungen heißen ***physikalisch ähnlich***, wenn alle Kennzahlen übereinstimmen. Sind nur einige Kennzahlen gleich, spricht man von ***partieller Ähnlichkeit***.

Die Fortpflanzungsgeschwindigkeit kleiner Druckstörungen heißt ***Schallgeschwindigkeit*** a. Für sie gilt:

$$a = \sqrt{\left(\frac{dp}{d\rho}\right)_s}, \tag{2.6}$$

wobei der Index s besagt, dass die Ableitung des Drucks nach der Dichte bei konstanter Entropie s erfolgen muss. Ferner kann noch eine wichtige Kennzahl eingeführt werden, die besonderes bei Strömungen mit hohen Geschwindigkeiten benutzt wird,

Name	Definition
Reynolds-Zahl	$\mathrm{Re} = \frac{\rho \bar{u} l}{\eta} = \frac{\bar{u} l}{\nu}$
Froude-Zahl	$\mathrm{Fr} = \frac{\bar{u}}{\sqrt{gl}}$
Prandtl-Zahl	$\mathrm{Pr} = \frac{\eta c_p}{\lambda}$
Eckert-Zahl	$\mathrm{Ec} = \frac{\bar{u}^2}{c_c T}$
Isobare Dichteänderungs-Zahl	$\mathrm{K}_\rho = -\beta T$

Tabelle 2.1: Kennzahlen

wenn zusätzlich elastische Kräfte aufgrund der Kompressibilität des Fluids auftreten. Die Mach-Zahl hängt von a ab:

$$\mathrm{Ma} = \frac{\bar{u}}{a}. \tag{2.7}$$

Wenn $\mathrm{Ma} < 0,3$ ist, dann können die Strömungen als inkompressibel betrachtet werden. In der Literatur findet man noch andere Kennzahlen, die mit den hier angegebenen zusammenhängen.

Für zwei Grenzfälle sehr kleiner Reynolds-Zahlen (***schleichende Strömungen***) und sehr großer Reynolds-Zahlen (***Grenzschichtströmungen***) lassen sich asymptotische Näherungslösungen entwickeln.

2.2 Sonderfälle

Die ziemlich komplizierte allgemeine Form der Navier-Stokes-Gleichungen versucht man durch besondere Eigenschaften eines bestimmten Strömungsmodells zu vereinfachen. Nur in Einzelfällen gelingt es, die reduzierten Gleichungen analytisch zu lösen ([150], Kap. 5).

In vielen praktischen Anwendungen kommen häufig Strömungen vor, deren Temperaturunterschiede sehr gering sind und vernachlässigt werden können. Solche isoenergetischen Modelle (ohne Gleichung 2.3) sind der Schwerpunkt dieser Arbeit.

2.2.1 Inkompressible Strömung

Nicht nur für Flüssigkeiten, sondern auch für Gase mit $\mathrm{Ma} < 0,3$ kann die Dichte ρ als konstant angenommen werden. Dann reduziert sich die Kontinuitätsgleichung auf:

$$\nabla \cdot \boldsymbol{u} = 0. \tag{2.8}$$

Obwohl die Impulsgleichungen (2.2) nicht geändert werden, ist die Annahme hilfreich bei der numerischen Lösung.

2.2.2 Axialsymmetrische Strömung

In zirkularen Geometrien wird häufig angenommen, dass keine Rotationsveränderung von Strömungsgrößen existiert. Dann werden die Navier-Stokes-Gleichungen im zylindrischen Koordinatensystem $\{r, \varphi, z\}$ aufgeschrieben. Die Komponenten der Geschwindigkeit $\boldsymbol{u} = (u, v, w)^T$ heißen nun: u - radiale Geschwindigkeit; v tangentiale oder Wirbel-Geschwindigkeit; w - axiale Geschwindigkeit. Im kompressiblen Fall gehen die Impulsgleichungen über in

$$\begin{aligned}
\rho\left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial r} + \frac{v}{r}\frac{\partial u}{\partial \varphi} - \frac{v^2}{r} + w\frac{\partial u}{\partial z}\right) &= f_r - \frac{\partial p}{\partial r} + \frac{1}{r}\frac{\partial (r\tau_{rr})}{\partial r} + \frac{1}{r}\frac{\partial \tau_{r\varphi}}{\partial \varphi} + \frac{\partial \tau_{rz}}{\partial z} - \frac{\tau_{\varphi\varphi}}{r} \\
\rho\left(\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial r} + \frac{v}{r}\frac{\partial v}{\partial \varphi} + \frac{uv}{r} + w\frac{\partial v}{\partial z}\right) &= f_\varphi - \frac{1}{r}\frac{\partial p}{\partial \varphi} + \frac{1}{r^2}\frac{\partial (r^2\tau_{r\varphi})}{\partial r} + \frac{1}{r}\frac{\partial \tau_{\varphi\varphi}}{\partial \varphi} + \frac{\partial \tau_{\varphi z}}{\partial z} \\
\rho\left(\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial r} + \frac{v}{r}\frac{\partial w}{\partial \varphi} + w\frac{\partial w}{\partial z}\right) &= f_z - \frac{\partial p}{\partial z} + \frac{1}{r}\frac{\partial (r\tau_{rz})}{\partial r} + \frac{1}{r}\frac{\partial \tau_{\varphi z}}{\partial \varphi} + \frac{\partial \tau_{zz}}{\partial z}
\end{aligned} \tag{2.9}$$

und die Kontinuitätsgleichung in

$$\frac{\partial \rho}{\partial t} + \frac{1}{r}\frac{\partial (\rho r u)}{\partial r} + \frac{1}{r}\frac{\partial (\rho v)}{\partial \varphi} + \frac{\partial (\rho w)}{\partial z} = 0, \tag{2.10}$$

wobei die Komponenten des Spannungstensors $\tau = \tau_{ij}$ $(i, j \in \{r, \varphi, z\})$ nun die folgenden sind:

$$\begin{aligned}
\tau_{r\varphi} = \tau_{\varphi r} &= \eta\left(r\frac{\partial}{\partial r}\left(\frac{v}{r}\right) + \frac{1}{r}\frac{\partial u}{\partial \varphi}\right), & \tau_{rr} &= 2\eta\left(\frac{\partial u}{\partial r} - \frac{1}{3}\nabla\cdot\boldsymbol{u}\right), \\
\tau_{\varphi z} = \tau_{z\varphi} &= \eta\left(\frac{\partial v}{\partial z} + \frac{1}{r}\frac{\partial w}{\partial \varphi}\right), & \tau_{\varphi\varphi} &= 2\eta\left(\frac{1}{r}\frac{\partial v}{\partial \varphi} + \frac{u}{r} - \frac{1}{3}\nabla\cdot\boldsymbol{u}\right), \\
\tau_{zr} = \tau_{rz} &= \eta\left(\frac{\partial w}{\partial r} + \frac{\partial u}{\partial z}\right), & \tau_{zz} &= 2\eta\left(\frac{\partial w}{\partial z} - \frac{1}{3}\nabla\cdot\boldsymbol{u}\right).
\end{aligned} \tag{2.11}$$

Wenn also die drei Geschwindigkeitskomponenten unabhängig von φ sind, kann eine 3D-Strömung auf einem 2D-Gitter effizient simuliert werden.

2.2.3 Stokes-Gleichungen

In schleichenden Strömungen ($\mathrm{Re} \ll 1$) mit einer hohen Viskosität kann die nichtlineare Konvektion vernachlässigt werden. Die Navier-Stokes-Gleichungen gehen dann in ein lineares System über und werden Stokes-Gleichungen genannt:

$$\begin{aligned}
\frac{\partial \boldsymbol{u}}{\partial t} - \nu\Delta\boldsymbol{u} + \nabla p &= \boldsymbol{f}, \\
\nabla\cdot\boldsymbol{u} &= 0,
\end{aligned} \tag{2.12}$$

wobei $\nu = \frac{\eta}{\rho}$ die kinematische Viskosität ist. Diese Gleichungen kann man auch nach der Linearisierung der Navier-Stokes-Gleichungen bekommen. Sie werden deshalb oft als ein vereinfachtes Modell untersucht.

2.2.4 Euler-Gleichungen

Die Euler-Gleichungen beschreiben eine nichtviskose Strömung. Das entspricht dem Fall $\eta = 0$, d.h. die Komponenten des Spannungstensors (2.4) verschwinden. Die Impulsgleichungen (2.2) reduzieren sich zu:

$$\rho \frac{D\boldsymbol{u}}{Dt} = \boldsymbol{f} - \nabla p. \tag{2.13}$$

Solche Strömungen entstehen weit von einem festen Rand entfernt, wo die viskosen Effekte gering sind. Das Modell wird häufig für die Strömungssimulationen mit einer großen Mach-Zahl angewendet. Wenn Geschwindigkeit und Reynoldszahl sehr hoch sind, ist die Viskosität wichtig in einer dünnen Schicht neben dem Rand. Dazu gehören z.B. Verbrennungs-, Detonations- und Explosionsprozesse [42].

2.3 Randbedingungen

Die korrekte Formulierung der Randbedingungen für die Navier-Stokes-Gleichungen ist ein kompliziertes und in einigen Aspekten noch offenes Problem [70]. Wir fassen hier die bekanntesten und im praktischen Sinne wichtigsten Randbedingungen zusammen. Für deren Formulierung bezeichne u_n die Geschwindigkeitskomponente senkrecht zum Rand (in Normalenrichtung), u_t die Geschwindigkeitskomponente parallel zum Rand (in Tangentialrichtung) und $\frac{\partial u_n}{\partial n}$ bzw. $\frac{\partial u_t}{\partial n}$, $\frac{\partial u_n}{\partial \boldsymbol{t}}$ bzw. $\frac{\partial u_t}{\partial \boldsymbol{t}}$, die jeweilige Ableitung in Normalen- und Tangentialrichtung, wobei die Tangentialrichtung $\boldsymbol{t}$ zum Unterschied von der Zeit fettgedruckt wurde. Wir beschränken uns ausschließlich auf die Randbedingungen für die $\boldsymbol{u}$-p-Formulierung (in primitiven Variablen) der inkompressiblen und instationären Navier-Stokes-Gleichungen in einem Zeitintervall $[0, T]$ und einem offenen und beschränkten Gebiet Ω:

$$\begin{cases} \dfrac{\partial \boldsymbol{u}}{\partial t} - \nu \Delta \boldsymbol{u} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} + \nabla p = \boldsymbol{f} & \text{in } Q_T := (0, T] \times \Omega \\ \nabla \cdot \boldsymbol{u} = 0 & \text{in } Q_T. \end{cases} \tag{2.14}$$

Weitere mögliche Formulierungen durch die Einführung von Fluss- und Wirbelfunktionen und die zugehörigen Randbedingungen findet man z.B. bei Gresho und Sani [70] oder Quartapelle [133].

Haftbedingung (no-slip)

Das ist die grundlegende Bedingung für eine viskose Strömung an einem festen Rand. Sie lautet: kein Fluid dringt durch die Wand, und das Fluid haftet an der Wand, d.h. es gilt

$$u_n = 0, \qquad u_t = 0. \tag{2.15}$$

Rutschbedingung (free-slip) und Symmetrie

Es dringt kein Fluid durch die Wand. Im Gegensatz zur Haftbedingung gibt es jedoch entlang der Wand keine Reibungsverluste, d. h.

$$u_n = 0, \qquad \frac{\partial u_t}{\partial n} = 0. \tag{2.16}$$

Die Rutschbedingung wird insbesondere auch bei axialsymmetrischen Problemen angewendet. Legt man dabei an der Symmetrieachse eine Rutschbedingung fest, so kann man sich bei der Berechnung auf das halbe Gebiet beschränken.

Eine andere Version der Symmetrie-Bedingung ist gut für Sonderfälle geeignet, wenn die Strömung senkrecht zur Symmetrieachse fließt:

$$u_t = 0, \qquad 2\eta \frac{\partial u_n}{\partial n} = P. \tag{2.17}$$

Freie Oberfläche (traction)

Diese Randbedingung stellt eine Kräftebilanz dar:

$$\eta(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T) \cdot \boldsymbol{n} - p\boldsymbol{n} = \boldsymbol{g}, \tag{2.18}$$

wobei $\boldsymbol{g}$ eine auf die freie Fluidoberfläche wirkende und vorgegebene Kraft ist. Für eine ebene Strömungsfläche wird die Bedingung vereinfacht:

$$\eta \left(\frac{\partial \boldsymbol{u}}{\partial n} + \nabla u_n \right) - p\boldsymbol{n} = \boldsymbol{g}.$$

Ein-/Ausströmbedingung (inflow/outflow)

Beide Geschwindigkeitskomponenten sind fest vorgegeben (Dirichlet-Randbedingung), d.h.

$$u_n = u_n^0, \qquad u_t = u_t^0, \qquad u_n^0, u_t^0 \text{ gegeben.} \tag{2.19}$$

Statt der Geschwindigkeit kann auch der Fluss der Ein- oder Ausströmung angegeben werden:

$$\boldsymbol{n} \cdot (\rho \boldsymbol{u}\boldsymbol{u} - \tau) + p\boldsymbol{n} = f_n,$$

wobei f_n die normale Komponente des Flusses und $\boldsymbol{u}\boldsymbol{u}$ die Matrix $\{u_i u_j\}$, $i, j = \overline{1, d}$, bezeichnen. Für ebene Ränder gilt:

$$\rho u_n \boldsymbol{u} + p\boldsymbol{n} - \eta \left(\frac{\partial \boldsymbol{u}}{\partial n} + \nabla u_n \right) = f_n. \tag{2.20}$$

Freie Ausströmbedingung (outflow)

Das sind die schwierigsten, nicht eindeutig definierten und kaum theoretisch unterstützten Randbedingungen. Eine natürliche Version der Ausströmbedingung folgt aus (2.18), wenn $(\nabla \boldsymbol{u})^T$ weggelassen wird. In 2D ergibt sich in Normalenrichtung:

$$2\eta \frac{\partial u_n}{\partial n} - p = \boldsymbol{n} \cdot \boldsymbol{g} = g_n \tag{2.21}$$

und in die Tangentialrichtung:

$$\eta \left(\frac{\partial u_t}{\partial n} + \frac{\partial u_n}{\partial \boldsymbol{t}} \right) = g_t. \tag{2.22}$$

Analytisch hergeleitet sind sie kaum praktisch anwendbar, weil die äußere Kraft $\boldsymbol{g}$ normalerweise unbekannt ist. Stattdessen verwendet man häufig die folgenden Randbedingungen: Beide Normalenableitungen der Geschwindigkeit ändern sich nicht in der Richtung senkrecht zum Rand, d.h.

$$\frac{\partial u_n}{\partial n} = 0, \qquad \frac{\partial u_t}{\partial n} = 0. \tag{2.23}$$

Eine Alternative wurde von Taylor *et al.* [159] vorgeschlagen. Als erste Schätzung wird $g_n = 0,\ g_t = 0$ angenommen. Mit der berechneten Lösung wird die Kraft $\boldsymbol{g}$ aktualisiert und so lange iteriert bis Konvergenz eintritt. Ein Näherungswert für $\boldsymbol{g}$ kann auch von der vorhergehenden Zeititeration bei der instationären Problemen übernommen werden. Der bekannte FIDAP-Code zur numerischen Strömungssimulation nutzt eine ganz ähnliche Methode, die den viskosen Anteil von (2.21) ignoriert und einfach $|g_n| = -p$ in jeder Iteration oder in jedem Zeitschritt setzt. Die beiden letzten Methoden wurden analytisch nicht untersucht.

Periodische Randbedingung

Bei in einer Achsenrichtung periodischen Problemen mit der Periodlänge a (z.B. die Strömung über eine gewellte Oberfläche) kann man sich auf die Berechnung einer Periode beschränken. Die Geschwindigkeiten und der Druck am linken und rechten Gebietsrand müssen dann identisch sein, z.B.

$$u_n(0, y) = u_n(a, y), \qquad u_t(0, y) = u_t(a, y), \qquad p(0, y) = p(a, y) \tag{2.24}$$

(Periode in x-Richtung, $x = 0$: linker Gebietsrand, $x = a$: rechter Gebietsrand).

Sind an allen Rändern die Geschwindigkeiten selbst und nicht deren Normalableitungen vorgegeben, so muss zusätzlich das Randintegral über die Geschwindigkeiten senkrecht zum Rand Null sein, d.h.

$$\int_{\partial\Omega} \boldsymbol{u} \cdot \boldsymbol{n}\, ds = 0. \tag{2.25}$$

Druckrandbedingungen

Um Bedingungen für den Druck stellen zu können, muss ein modifiziertes System der Navier-Stokes-Gleichungen gelöst werden. Es besteht aus der Impulsgleichung und Druck-Poisson-Gleichung. Die letzte Gleichung wird durch die Anwendung des Divergenzoperators auf die Impulsgleichung mit der Berücksichtigung der Kontinuität $\nabla \cdot \boldsymbol{u} = 0$ hergeleitet. Dieses setzt voraus, dass die Lösung der $\boldsymbol{u} - p$-Formulierung (2.14) glatt genug ist, so dass die Anwendung des Divergenzoperators einen Sinn hat. Die endgültige Version sieht folgendermaßen aus:

$$\Delta p = \rho \nabla \cdot (\boldsymbol{f} - (\boldsymbol{u} \cdot \nabla)\boldsymbol{u}), \tag{2.26}$$

die allerdings im Vergleich zu (2.14) zur nicht eindeutigen Lösung des Navier-Stokes-Problems führen kann (siehe [70], Abschnitt 3.10.3). Die bessere Formulierung behält den viskosen Term bei:

$$\Delta p = \rho \nabla \cdot (\boldsymbol{f} + \nu \Delta \boldsymbol{u} - (\boldsymbol{u} \cdot \nabla)\boldsymbol{u}). \tag{2.27}$$

Die Navier-Stokes-Gleichungen bestehend aus der Impulsgleichung und der Druck-Poisson-Gleichung können genau so, wie die übliche $\boldsymbol{u} - p$-Formulierung benutzt werden. Das Geschwindigkeitsfeld der Lösung ist dabei immer divergenzfrei.

Wenn die Normalenkomponente der Geschwindigkeit $u_n = \boldsymbol{n} \cdot \boldsymbol{u}$ auf dem Rand $\partial\Omega$ gegeben ist, dann sieht die korrekte Neumann-Randbedingung für den Druck wie folgt aus:

$$\frac{\partial p}{\partial n} = \rho \boldsymbol{n} \cdot \left(\boldsymbol{f} + \nu \Delta \boldsymbol{u} - (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} - \frac{\partial \boldsymbol{u}}{\partial t} \right) \qquad \text{auf } \partial\Omega, \tag{2.28}$$

wobei der Druck bis auf eine Konstante bestimmt wird.

2.4 Anfangsbedingungen

Für eine instationäre Strömung müssen auch Anfangsbedingungen gestellt werden. Außer gegebenen Dirichlet-Bedingungen auf Randteilen, muss die Geschwindigkeit $\boldsymbol{u}_0$ überall in Ω der Inkompressibilitäts-Bedingung genügen ([70], Abschnitt 3.9):

$$\nabla \cdot \boldsymbol{u}_0 = 0 \qquad \text{in } \Omega. \tag{2.29}$$

Es gibt keine Anfangsbedingungen für den Druck. Das Anfangsdruckfeld kann durch die Druck-Poisson-Gleichung (2.27) und die Neumann-Randbedingung (2.28) eingesetzt werden ([82]; [70], Abschnitt 3.9.2).

Kapitel 3

Numerische Approximation

Denken heißt vergleichen.
- Walther Rathenau

Die instationären Navier-Stokes-Gleichungen werden in diesem Kapitel unabhängig voneinander bzgl. der Zeit und des Raumes diskretisiert. Implizite Methoden mit Druck-Geschwindigkeits-Kopplung werden bevorzugt. Der Schwerpunkt der Untersuchung besteht in der Reformulierung der Galerkinschen Finite-Elemente-Methode (FEM) zu den konservativen Finite-Volumen-Methoden (FVM), um wichtige theoretische Eigenschaften der GFEM dadurch für die FVM zu gewinnen.

3.1 Zeitdiskretisierung und Linearisierung

3.1.1 Finite-Differenzen-Approximation

Wir betrachten die zeitabhängigen inkompressiblen Navier-Stokes-Gleichungen auf einem Zeitintervall $[0,T]$ und einem offenen und beschränkten Gebiet Ω von $\mathbb{R}^d\,(d=2,3)$ mit dem Rand $\partial\Omega$ in der Form:

$$\begin{cases} \dfrac{\partial \boldsymbol{u}}{\partial t} - \nu\Delta\boldsymbol{u} + (\boldsymbol{u}\cdot\nabla)\boldsymbol{u} + \nabla p = \boldsymbol{f} & \text{in } Q_T := (0,T]\times\Omega \\ \nabla\cdot\boldsymbol{u} = 0 & \text{in } Q_T \\ \boldsymbol{u} = 0 & \text{auf } (0,T]\times\partial\Omega \\ \boldsymbol{u}_{|t=0} = \boldsymbol{u}_0 & \text{auf } \Omega, \end{cases} \tag{3.1}$$

wobei $\boldsymbol{f} = \boldsymbol{f}(t,\boldsymbol{x})$ und $\boldsymbol{u}_0 = \boldsymbol{u}_0(\boldsymbol{x})$ gegebene Funktionen sind. Zur Abkürzung werden die Randbedingungen in homogener Form aufgeschrieben, weil sie in diesem Kapitel nicht näher betrachtet werden.

Das nichtlineare Problem wird zuerst in der Zeit diskretisiert und linearisiert und dann zu jedem Zeitpunkt im Raum diskretisiert. Sei $t_n = n\triangle t$. Dann kann die Zeit in der Gleichung (3.1_1) durch Finite-Differenzen (Fractional-Step-Methoden) oder mit Hilfe der Projektions-Methoden (Nonfractional-Step-Methoden) approximiert werden. Aus der ersten Klasse wird das Einschritt-θ-Schema bevorzugt [41, 137].

Mit den Näherungen

$$\begin{aligned} \boldsymbol{u} &\approx \theta \boldsymbol{u}^{n+1} + (1-\theta)\boldsymbol{u}^n, \\ p &\approx \theta p^{n+1} + (1-\theta)p^n, \qquad \text{für } \theta \in [0,1] \\ \boldsymbol{f} &\approx \theta \boldsymbol{f}^{n+1} + (1-\theta)\boldsymbol{f}^n \end{aligned}$$

und mit der Wahl von θ können unterschiedliche Einschritt-Verfahren bezüglich der Zeit t gewonnen werden.

$$\begin{cases} \dfrac{\boldsymbol{u}^{n+1}-\boldsymbol{u}^n}{\triangle t} - \nu\Delta(\theta\boldsymbol{u}^{n+1} + (1-\theta)\boldsymbol{u}^n) + \theta\boldsymbol{u}^{n+1}\cdot\nabla\boldsymbol{u}^{n+1} + (1-\theta)\boldsymbol{u}^n\cdot\nabla\boldsymbol{u}^n \\ \qquad\qquad +\nabla(\theta p^{n+1} + (1-\theta)p^n) = (\theta\boldsymbol{f}^{n+1} + (1-\theta)\boldsymbol{f}^n), \\ \nabla\cdot\boldsymbol{u}^{n+1} = 0. \end{cases} \tag{3.2}$$

Das resultierende nichtlineare Randwertproblem (3.2) bleibt immer implizit wegen der Kontinuitätsgleichung (3.2_2), sogar wenn $\theta = 0$ gewählt wird. Der nichtlineare Konvektionsterm kann in der Umgebung von $(\boldsymbol{u}^{n+1}, p^{n+1})$ durch eine NEWTONsche Iteration linearisiert werden. So ergibt sich das implizite EULER-Schema ($\theta = 1$):

$$\frac{\boldsymbol{u}^{n+1}-\boldsymbol{u}^n}{\triangle t} - \nu\Delta\boldsymbol{u}^{n+1} + \boldsymbol{u}^n\cdot\nabla\boldsymbol{u}^{n+1} + \boldsymbol{u}^{n+1}\cdot\nabla\boldsymbol{u}^n - \boldsymbol{u}^n\cdot\nabla\boldsymbol{u}^n + \nabla p^{n+1} = \boldsymbol{f}^{n+1}. \tag{3.3}$$

Die Annahme $\boldsymbol{u}^{n+1}\cdot\nabla\boldsymbol{u}^n \approx \boldsymbol{u}^n\cdot\nabla\boldsymbol{u}^n$ liefert die semi-explizite EULER-Methode:

$$\frac{\boldsymbol{u}^{n+1}-\boldsymbol{u}^n}{\triangle t} - \nu\Delta\boldsymbol{u}^{n+1} + \boldsymbol{u}^n\cdot\nabla\boldsymbol{u}^{n+1} + \nabla p^{n+1} = \boldsymbol{f}^{n+1}, \tag{3.4}$$

die eine Methode erster Ordnung bezüglich $\triangle t$ ist. Die zweite Ordnung des θ-Schemas kann nur bei $\theta = \frac{1}{2}$ (Trapezregel) erreicht werden. Dann erhält man mit der Approximation des konvektiven Terms durch zwei vorhergehenden Zeitschritte die bekannte CRANK-NICOLSON/ADAMS-BASHFORTH-Methode:

$$\frac{2(\boldsymbol{u}^{n+1}-\boldsymbol{u}^n)}{\triangle t} - \nu(\Delta\boldsymbol{u}^{n+1} + \Delta\boldsymbol{u}^n) + 3\boldsymbol{u}^n\cdot\nabla\boldsymbol{u}^n - \boldsymbol{u}^{n-1}\cdot\nabla\boldsymbol{u}^{n-1} + \nabla p^{n+1} + \nabla p^n = \boldsymbol{f}^{n+1} + \boldsymbol{f}^n, \tag{3.5}$$

wobei der nichtlineare konvektive Term schon vollständig explizit erscheint. Das letztgenannte Schema kann auch aus dem Prädiktor-Korrektor-Schema hergeleitet werden und ist unbedingt stabil (siehe Marchuk [104]).

3.1.2 Projektionsmethoden

Eine andere Art von Zeitdiskretisierungs-Methoden sind die sog. Projektionsmethoden, die zuerst von Chorin (1967) und Temam (1969) vorgeschlagen wurden [137, 138, 160]. Die Idee besteht darin, dass die Geschwindigkeit $\boldsymbol{u}$ und der Druck p getrennt berechnet werden können. Dabei wird eine Zwischengeschwindigkeit errechnet, die dann auf den Unterraum der divergenzfreien Funktionen projiziert wird. Die theoretische Grundlage dazu geben das HELMHOLTZ-Zerlegungsprinzip und der

Satz von LADYZHENSKAJA [68]. Wir definieren zuerst den Raum der divergenzfreien Funktionen

$$H_{\text{div}} := \left\{ \boldsymbol{v} \in (L^2(\Omega))^d \mid \nabla \cdot \boldsymbol{v} = 0 \text{ in } \Omega, \boldsymbol{v} \cdot \boldsymbol{n} = 0 \text{ auf } \partial\Omega \right\}. \tag{3.6}$$

Das Zerlegungsprinzip lautet: eine beliebige Vektorfunktion $\boldsymbol{v} \in (L^2(\Omega))^d$ kann eindeutig als Summe $\boldsymbol{v} = \boldsymbol{w} + \nabla\psi$ dargestellt werden, wobei $\boldsymbol{w} \in H_{\text{div}}$ und $\psi \in H^1(\Omega)$ sind.

Die klassische Chorin-Temam-Methode besteht aus den nächsten zwei Schritten:

1. Die Zwischengeschwindigkeit $\boldsymbol{u}^{n+1/2}$ wird als Lösung des folgenden nichtlinearen Konvektions-Diffusions-Problems

$$\begin{cases} \dfrac{\boldsymbol{u}^{n+1/2} - \boldsymbol{u}^n}{\triangle t} - \nu\Delta\boldsymbol{u}^{n+1/2} + (\boldsymbol{u}^{n+1/2} \cdot \nabla)\boldsymbol{u}^{n+1/2} \\ \qquad\qquad + \dfrac{1}{2}(\nabla \cdot \boldsymbol{u}^{n+1/2})\boldsymbol{u}^{n+1/2} = \boldsymbol{f}^{n+1/2} & \text{in } \Omega \\ \boldsymbol{u}^{n+1/2} = 0 & \text{auf } \partial\Omega \end{cases} \tag{3.7}$$

   bestimmt. Der Ausdruck $\frac{1}{2}(\nabla \cdot \boldsymbol{u}^{n+1/2})\boldsymbol{u}^{n+1/2}$ wird aus Stabilisierungsgründen eingeführt.

2. Die Größen $\boldsymbol{u}^{n+1}$ und q^{n+1} sind mit Hilfe des folgenden Problems zu bestimmen:

$$\begin{cases} \dfrac{\boldsymbol{u}^{n+1} - \boldsymbol{u}^{n+1/2}}{\triangle t} + \nabla q^{n+1} = 0 & \text{in } \Omega, \\ \nabla \cdot \boldsymbol{u}^{n+1} = 0 & \text{in } \Omega, \\ \boldsymbol{u}^{n+1} \cdot \boldsymbol{n} = 0 & \text{auf } \partial\Omega, \end{cases} \tag{3.8}$$

   wobei $\boldsymbol{n}$ der äußere Normalvektor auf $\partial\Omega$ ist. Das System (3.8) entsteht, indem $\boldsymbol{u}^{n+1}$ als die L^2-orthogonale Projektion von $\boldsymbol{u}^{n+1/2}$ auf dem Raum H_{div} gewählt wird und der Skalar q^{n+1} aus dem Helmholtz-Zerlegungsprinzip folgt ($\boldsymbol{v} = \boldsymbol{u}^{n+1/2}$, $\boldsymbol{w} = \boldsymbol{u}^{n+1}$ und $\nabla\psi = \triangle t \nabla q^{n+1}$). Eine Anwendung des Divergenzoperators auf (3.8) liefert dann das Neumann-Problem für q^{n+1}

$$\begin{cases} \Delta q^{n+1} = \dfrac{1}{\triangle t}\nabla \cdot \boldsymbol{u}^{n+1/2} & \text{in } \Omega, \\ \dfrac{\partial q^{n+1}}{\partial \boldsymbol{n}} = 0 & \text{auf } \partial\Omega \end{cases} \tag{3.9}$$

   zusammen mit der letzten Korrektur für das Geschwindigkeitsfeld

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^{n+1/2} - \triangle t \nabla q^{n+1}. \tag{3.10}$$

Die Größe $q^{n+1}(\boldsymbol{x})$ approximiert den physikalischen Druck $p(t_{n+1}, \boldsymbol{x})$ nicht ausreichend gut [137], weil der Druck p ein Poisson-Problem (2.27) mit inhomogenen Neumann-Randbedingungen (2.28) erfüllen muss, wobei das Problem (3.9) die homogenen Neumann-Randbedingungen enthält. Eine dafür geeignete Modifikation der

Projektions-Methode sowie ein Projektionsschema höhere Ordnung bzgl. der Zeit findet man bei Gresho [71, 72]. Die Projektionsmethode wird trotzdem sehr breit mit der Finite-Elemente- und mit Spektral-Methoden zusammen verwendet (Beispiele sind die Programmpakete [40] und [108]). Eine weitere Entwicklung hat Perot [130] durchgeführt und gezeigt, wenn eine Aufgabe vollständig diskretisiert und als ein lineares Gleichungssystem dargestellt ist, dann ist die Projektions-Methode zu einer Block-LU-Zerlegung dieses Systems äquivalent. Die Fortsetzung dieser Idee wurde von Quarteroni [134, 136] als unterschiedliche Faktorisierungsmethoden gefunden.

Zusammenfassung

Eine Anwendung der Zeitdiskretisierung und der Linearisierung z.B. durch das Crank-Nicolson/Adams-Bashforth-Schema reduziert die Navier-Stokes-Gleichungen auf das Oseen- oder Stokes-Problem [50, 51]. Letzteres in der Form

$$\begin{cases} -\nu\Delta \boldsymbol{u} + \nabla p = \boldsymbol{f} & \text{in } \Omega \\ \nabla \cdot \boldsymbol{u} = 0 & \text{in } \Omega \end{cases} \tag{3.11}$$

betrachten wir in den nächsten Abschnitten und diskretisieren es mit Hilfe von Finite-Volumen- (FVM) und Finite-Elemente-Methoden (FEM).

Außer den zwei betrachteten Strategien existieren auch andere weniger bekannte Methoden (Quarteroni und Valli [137], S.447 und weitere Referenzen dort), die aber im Kontext mit den spektralen Methoden angewendet werden, die außerhalb der Betrachtungen dieser Arbeit liegen.

3.2 Raumdiskretisierung

Wir interessieren uns für unstrukturierte Dreieck- oder Viereckgitter, deren Eigenschaften die Grundlage für die Anwendung der Finite-Elemente- oder der Finite-Volumen-Methode sind. Eine Delaunay-Triangulierung wird im Zusammenhang mit dem Voronoi- und dem Donald-Diagramm betrachtet, die sehr häufig bei den praktischen Aufgaben verwendet werden.

Das Thema berührt sofort die Frage der Gittergenerierung im Gebiet Ω. Eine ausführliche Theorie dazu findet man bei Knupp [92], Liseikin [99] für strukturierte Gitter und bei Carey [27], George [66, 65] für unstrukturierte Gitter. Die Besonderheiten der Gittergestalt in Abhängigkeit von der Gebietsgeometrie findet man in [162]). Wir beschränken uns auf Definitionen und bestimmte Eigenschaften einer im Gebiet Ω existierenden Gitter $\mathcal{T}_h$.

Sei $\Omega \in \mathbb{R}^d$ $(d = 2, 3)$ offen und beschränkt. Ein Gitter $\mathcal{T}_h$ besteht aus Dreiecken oder Vierecken (Tetraeder oder Hexaeder) $K_i (i = 1, ..., N_T)$ so, dass $\Omega = \mathcal{T}_h = \bigcup_i K_i$ erfüllt ist und beliebige zwei K_i und K_j $(i \neq j)$ keinen gemeinsamen inneren Punkt besitzen. Wenn noch die Menge $K_i \cap K_j \neq \emptyset$ $(i \neq j)$ ein Knoten oder eine Kante ist, dann heißt das Gitter ***konform***. Ein nichtkonformes Gitter kann z.B. bei einer Adaption mit hängenden Knoten auftreten. Für eine konforme Triangulierung gilt: $N_T = 2N_K - N_R - d$, wobei N_K und N_R die gesamte Anzahl der Knoten und Anzahl der Randknoten entsprechend bezeichnen. Das Maß h ist die Länge der größten Kante und kennzeichnet die Feinheit des Gitters

$$h := \max\{h_K = \operatorname{diam}(K_i) \mid K_i \in \mathcal{T}_h\}. \tag{3.12}$$

Definition 3.1. *Eine Triangulierung $\mathcal{T}_h$ heißt Delaunay-Triangulierung, wenn der Umkreis jedes Dreiecks (Tetraeders) $K_i\,(i = 1, ..., N_T)$ keinen Knoten eines anderen Dreiecks (Tetraeders) enthält (z.B. Abb. 3.1a,b). Falls der Knoten auf dem Umkreis liegt, ist die Delaunay-Triangulierung nicht eindeutig (Abb. 3.1c). Die Kanten der Triangulierung werden auch Delaunay-Kanten genannt.*

Da die Summe von gegenüberliegenden Winkeln angrenzender Dreiecke mit gemeinsamer Kante in einem Umkreis (Abb. 3.1,c) gleich π ist, kann man behaupten, dass eine Triangulierung mit nicht stumpfwinkligen Dreiecken eine Delaunay-Triangulierung bildet. Die inverse Aussage stimmt nicht, weil eine allgemeine Delaunay-Triangulierung stumpfwinklige Dreiecke zulässt (Abb. 3.1,b). Das Umkreiszentrum eines solchen Dreiecks liegt außerhalb des Dreiecks.

Ein Voronoi-Diagramm (auch Dirichlet-Mosaik) ist eine grafisch duale geometrische Struktur zur Delaunay-Triangulierung und kann als eine andere Gebietsdiskretisierung aufgefasst werden. Das Voronoi-Diagramm kann auf der Grundlage einer Delaunay-Triangulierung eindeutig definiert werden. Diese Aussage gilt auch umgekehrt [8].

Definition 3.2. *Sei $S = \{A_i \mid i = 1, ..., N_K\}$ die Menge aller Ecken der Triangulierung $\mathcal{T}_h$. Der* ***Bisektor*** *einer Kante A_iA_j, $A_i, A_j \in S$, $(i \neq j)$ ist eine Mittelsenkrechtlinie (Mittelsenkrechtfläche) zu dieser Kante, die eine den Punkt A_i enthaltende Halbebene (Halbraum) definiert:*

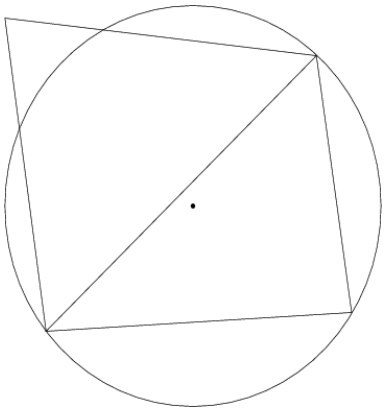
$$D(A_i, A_j) = \{x \mid d(A_i, x) < d(A_j, x)\}\,,$$

wobei $d(\cdot, \cdot)$ der Euklidische Abstand ist. Der Durchschnitt solcher Halbebenen (Halbräume) bildet das Voronoi-Polygon $\mathcal{V}_i$ des Punktes A_i, das eine Einheit des Voronoi-Diagramms $\mathcal{V}$ der Menge S ist:

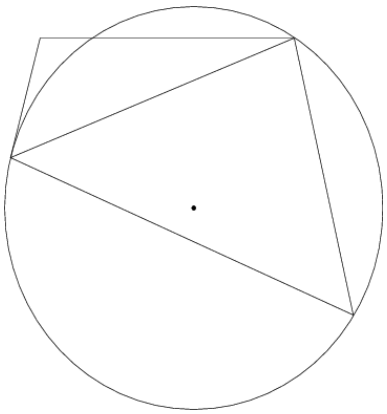
$$\mathcal{V}_i = \bigcap_{\substack{j=1 \\ j\neq i}}^{m} D(A_i, A_j), \qquad \mathcal{V} = \bigcup_{\substack{i,j=1 \\ j\neq i}}^{m} \mathcal{V}_i \cap \mathcal{V}_j. \tag{3.13}$$

Der Rand zwischen zwei Voronoi-Polygonen heißt Voronoi-Kante (oder -Fläche in 3D), wenn er mehr als einen Punkt enthält. Dabei werden Endpunkte einer Voronoi-Kante auch Voronoi-Punkte genannt.

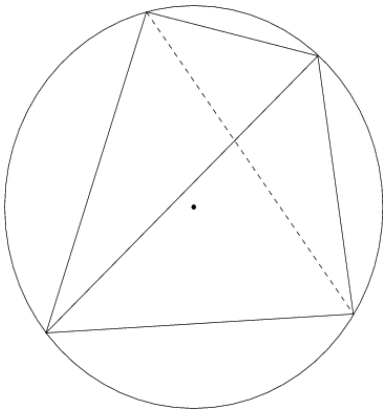
Nach Definition ist ein Voronoi-Polygon offen und konvex, aber nicht beschränkt am Rand des Gebietes (Abb. 3.2a, 3.3a). Alle Voronoi-Polygone sind disjunkt, wobei zwei benachbarte Polygone aus der Konvexität nur eine gemeinsame Kante (Fläche in 3D) belegen. Es ist nun offensichtlich, dass die Voronoi-Punkte mit den Umkreiszentren der Delaunay-Triangulierung übereinstimmen. In einem Voronoi-Punkt stoßen Randstücke von so vielen Voronoi-Polygonen zusammen, wieviel Knoten auf dem Umkreis liegen, d.h. mindestens drei (vier in 3D) Voronoi-Polygone. Ein ***regulärer*** Voronoi-Punkt gehört genau zu drei (vier in 3D) Voronoi-Polygonen und entspricht dem Fall (a) auf der Abbildung 3.1, wohingegen ein ***degenerierter*** Voronoi-Punkt wenigstens vier (fünf in 3D) Voronoi-Polygone gemeinsam hat (Abb. 3.1c). In diesem Fall treffen zwei oder mehr Dreieck-Umkreiszentren in einem Voronoi-Punkt zusammen. Im Fall einer Nicht-Delaunay-Triangulierung (Abb. 3.1d) kreuzen sich



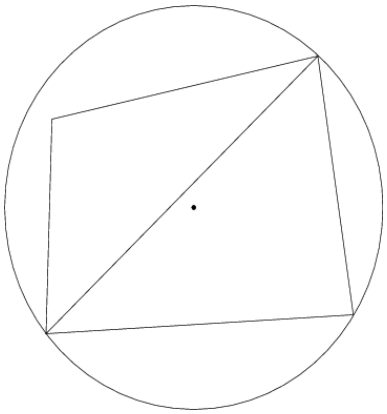
a) Delaunay-Triangulierung



b) Delaunay-Triangulierung mit einem stumpfen Dreieck



c) Nicht eindeutige Delaunay-Triangulierung



d) Nicht-Delaunay-Triangulierung

Abbildung 3.1: Delaunay- und Nicht-Delaunay-Triangulierung

die Mittelsenkrechtlinien der zwei benachbarten Dreiecke außerhalb eines Umkreiszentrums und können dadurch kein Polygon bilden.

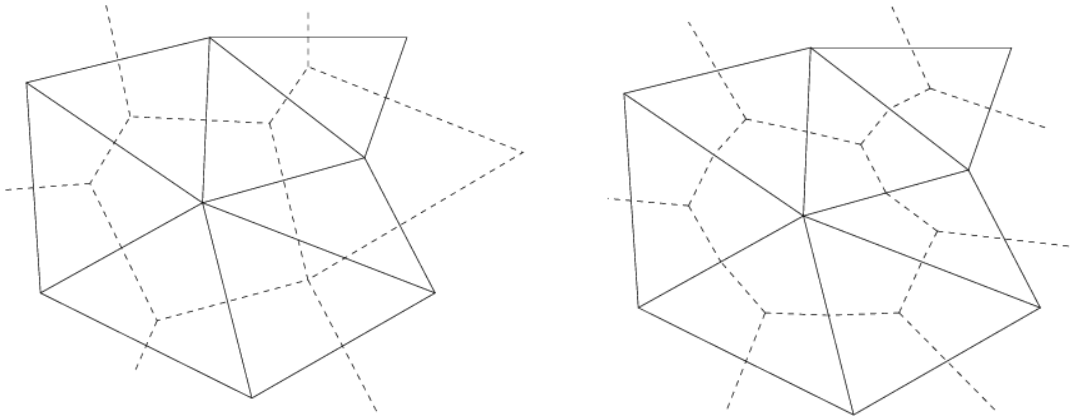
Das Voronoi-Diagramm ist eine optimale Diskretisierung, die Umgebungen der Knoten A_i $(i = 1, ..., N_K)$ darstellt, weil es für $m \geq 3$ nur $O(m)$ Kanten (Flächen in 3D) und Punkte gibt. Die durchschnittliche Anzahl von Kanten eines Voronoi-Polygons in 2D ist immer kleiner als 6 (bewiesen als Lemma 2.3 in [8]).

Wir betrachten nun ein Dreieck K_i einer Delaunay-Triangulierung $\mathcal{T}$ mit den Ecken $A_1(x_1, y_1), A_2(x_2, y_2), A_3(x_3, y_3)$. Dann kann das entsprechende Umkreiszentrum bzw. ein Voronoi-Punkt $A_V(x_V, y_V)$ aus der Eigenschaft

$$|A_V A_1| = |A_V A_2| = |A_V A_3|$$

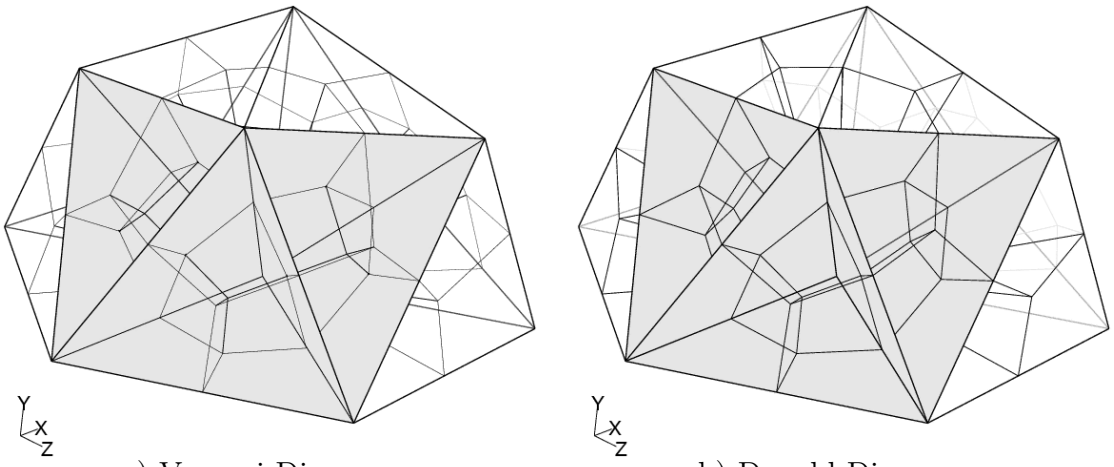
folgendermaßen

$$x_V = \frac{\begin{vmatrix} x_1^2+y_1^2 & x_2^2+y_2^2 & x_3^2+y_3^2 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{vmatrix}}{\begin{vmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{vmatrix}}, \quad y_V = -\frac{\begin{vmatrix} x_1^2+y_1^2 & x_2^2+y_2^2 & x_3^2+y_3^2 \\ x_1 & x_2 & x_3 \\ 1 & 1 & 1 \end{vmatrix}}{\begin{vmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{vmatrix}}$$



a) Voronoi-Diagramm b) Donald-Diagramm

Abbildung 3.2: Delaunay-Triangulierung (ganze Linie) und das grafisch duale a) Voronoi- und b) Donald-Diagramm (gestrichelte Linie)



a) Voronoi-Diagramm b) Donald-Diagramm

Abbildung 3.3: Dreidimensionale Voronoi- und Donald-Diagramme im Vergleich

oder effizienterweise (nur 20 Produkte) errechnet werden:

$$\begin{aligned} x_V &= \frac{[(x_1-x_3)(x_1+x_3)+(y_1-y_3)(y_1+y_3)](y_2-y_3)-[(x_2-x_3)(x_2+x_3)+(y_2-y_3)(y_2+y_3)](y_1-y_3)}{2[(x_1-x_3)(y_2-y_3)-(x_2-x_3)(y_1-y_3)]}, \\ y_V &= \frac{[(x_2-x_3)(x_2+x_3)+(y_2-y_3)(y_2+y_3)](x_1-x_3)-[(x_1-x_3)(x_1+x_3)+(y_1-y_3)(y_1+y_3)](x_2-x_3)}{2[(x_1-x_3)(y_2-y_3)-(x_2-x_3)(y_1-y_3)]}. \end{aligned} \tag{3.14}$$

Eine Alternative zum Voronoi-Diagramm ist das Donald-Diagramm, das die Punktmenge der Dreieck-Schwerpunkte als Ausgangspunkt hat (Abb. 3.2b, 3.3b). Daraus folgt direkt, dass die Triangulierung nicht unbedingt vom Delaunay-Typ sein muss, da ein Schwerpunkt immer innerhalb seines Dreiecks liegt und zusammen mit den Kanten-Mittelpunkten zwei Polygonkanten bildet. Die dabei entstehenden Polygone besitzen normalerweise Ecken an diesen Kanten-Mittelpunkten. Deshalb ist ein Donald-Polygon im Allgemeinen nicht konvex und die durchschnittliche Anzahl seiner Kanten in 2D ist kleiner als 12 (das Doppelte eines Voronoi-Polygons). Die

kartesischen Koordinaten eines Schwerpunktes lassen sich aber durch die Formeln

$$\begin{aligned} x_S &= \tfrac{1}{3}\left(x_1 + x_2 + x_3\right), \\ y_S &= \tfrac{1}{3}\left(y_1 + y_2 + y_3\right) \end{aligned} \tag{3.15}$$

mit wesentlich kleinerem Rechenaufwand im Vergleich zu einem Voronoi-Punkt (3.14) ausrechnen. Es gibt auch eine mittlere Alternative von Friedrich [57], die Dreieckzentren modifiziert ausrechnet:

$$\begin{aligned} x_S &= \sum_{i=1}^{3} \alpha_i x_i \\ y_S &= \sum_{i=1}^{3} \alpha_i y_i \end{aligned} \quad , \quad \alpha_i = \frac{\sum\limits_{j=1,\, j\neq i}^{3} l_j}{2\sum\limits_{j=1,}^{3} l_j}, \quad \begin{aligned} &l_j = |a_i - a_k|, \\ &i,j,k \in \{1,2,3\},\ i \neq j \neq k \end{aligned} . \tag{3.16}$$

Diese Definition führt zu einer viel besseren Approximation des Kontrolvolumens für verzerrte Dreiecke als die Formel (3.15). Die Formeln (3.14-3.16) können ohne Schwierigkeiten auf den dreidimensionalen Fall übertragen werden. Es kann allerdings grafisch nicht mehr übersichtlich dargestellt werden. Die Abbildung (3.3) ist eine grobe 3D-Triangulierung einer Halbkugel mit dem entsprechenden Voronoi- und Donald-Diagramm.

3.3 Finite-Volumen-Methoden

Die Finite-Volumen-Methoden wurden insbesondere für Erhaltungsgesetze entworfen, da sie für die Erhaltung der Masse und weiterer Bilanzvariablen in jedem Kontrollvolumen sorgen. Sie wurden zum ersten Mal von McDonald (1971) und MacCormack und Paullay (1972) für zweidimensionale Strömungssimulationen eingeführt und später von Rizzi und Inouye (1973) auf den dreidimensionalen Fall erweitert.

Wir multiplizieren das Stokes-Problem (3.11) mit einer Funktion $\boldsymbol{\varphi}$, integrieren das Resultat über das Gebiet Ω und verwenden die Green-Formel

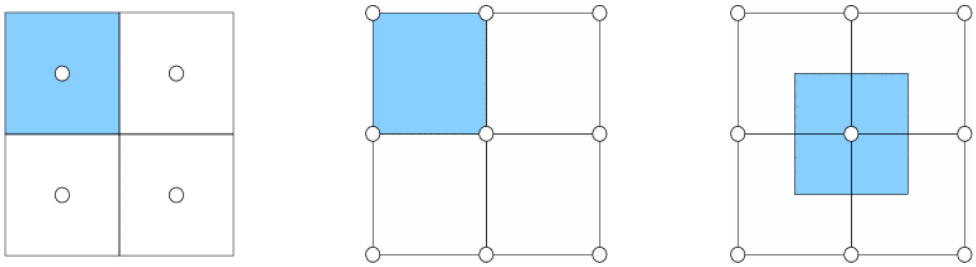
$$\begin{cases} \nu \displaystyle\int_\Omega \nabla \boldsymbol{u} \cdot \nabla \boldsymbol{\varphi} d\boldsymbol{x} - \nu \int_{\partial\Omega} \boldsymbol{\varphi} \nabla \boldsymbol{u} \cdot \boldsymbol{n} ds - \int_\Omega p \nabla \cdot \boldsymbol{\varphi} d\boldsymbol{x} + \int_{\partial\Omega} p\boldsymbol{\varphi} \cdot \boldsymbol{n} ds = \int_\Omega \boldsymbol{f}\boldsymbol{\varphi} d\boldsymbol{x}, \\ \displaystyle\int_{\partial\Omega} \boldsymbol{u} \cdot \boldsymbol{n} ds = 0, \end{cases} \tag{3.17}$$

wobei $\boldsymbol{n}$ der äußere Einheits-Normalvektor ist. Falls die Funktion $\boldsymbol{\varphi}$ auf dem Rand $\partial\Omega$ gleich Null ist, verschwinden die Randintegrale; die Gleichung (3.17) wird in die Finite-Elementen-Formulierung überführt, die im nächsten Abschnitt beschrieben wird. Wenn die Funktion $\boldsymbol{\varphi}$ stückweise konstant auf einem Kontrollvolumen Ω_i ist und sonst im Gebiet Ω verschwindet, dann zerfällt (3.17) in die Gleichungen für jedes Kontrollvolumen Ω_i

$$\begin{cases} -\nu \displaystyle\int_{\partial\Omega_i} \boldsymbol{\varphi} \nabla \boldsymbol{u} \cdot \boldsymbol{n} ds + \int_{\partial\Omega_i} p\boldsymbol{\varphi} \cdot \boldsymbol{n} ds = \int_{\Omega_i} \boldsymbol{f}\boldsymbol{\varphi} d\boldsymbol{x}, \\ \displaystyle\int_{\partial\Omega_i} \boldsymbol{u} \cdot \boldsymbol{n} ds = 0. \end{cases} \tag{3.18}$$

Die von $\boldsymbol{\varphi} = \mathbf{1}$ verschiedene Wahl der stückweise konstanten Funktion $\boldsymbol{\varphi}$ im Volumen Ω_i definiert die gewichtete Finite-Volumen-Formulierung. Die Methode (3.18) kann auch als die Petrov-Galerkinsche FEM betrachtet werden, wenn die bestimmten Basisfunktionen für die Geschwindigkeit $\boldsymbol{u}$ genommen werden.

Wir unterscheiden drei Arten der FVM: ***cell-centered-***, ***cell-vertex-*** und ***node-centered***-Schema nach der Lage der Unbekannten bezüglich der Kontrollvolumina (Abb. 3.4).



a) Cell-centered b) Cell-vertex c) Node-centered

Abbildung 3.4: Drei Arten der FVM

3.4 Finite-Elemente-Methoden

3.4.1 Variationsformulierung

Als Grundprinzip der FEM wird eine schwache Lösung zu dem Problem (3.1) gesucht. Neben dem eingeführten Raum H_{div} (3.6) werden noch folgende Räume definiert:

$$
\begin{aligned}
&H_0^1(\Omega) := \{u \in H^1(\Omega) \,|\, u = 0 \text{ auf } \partial\Omega\}\,,\\
&L_0^2(\Omega) := \left\{q \in L^2(\Omega) \,|\, \textstyle\int_\Omega q = 0\right\},\\
&V := \left(H_0^1(\Omega)\right)^d, \qquad Q := L_0^2(\Omega),\\
&V_{\text{div}} := \{\boldsymbol{v} \in V \,|\, \nabla \cdot \boldsymbol{v} = 0\}.
\end{aligned}
$$

Der effektivste Weg, das instationäre Navier-Stokes-Problem (3.1) zu lösen, ist ein stationäres Navier-Stokes-Problem in jedem Zeitschritt mittels finiter Differenzen-Approximation in der Zeit (Abschnitt 3.1.1) zu betrachten. Die Variationsformulierung für das stationäre Navier-Stokes-Problem lautet: es sind die Unbekannten $\boldsymbol{u} \in V$ und $p \in Q$ aus dem folgenden System zu finden:

$$
\begin{cases}
a(\boldsymbol{u}, \boldsymbol{v}) + c(\boldsymbol{u}; \boldsymbol{u}, \boldsymbol{v}) + b(\boldsymbol{v}, p) = (\boldsymbol{f}, \boldsymbol{v}) & \forall \boldsymbol{v} \in V\\
b(\boldsymbol{u}, q) = 0 & \forall q \in Q
\end{cases}, \tag{3.19}
$$

wobei die Bilinearformen a, b und die Trilinearform c wie folgt definiert sind:

$$\begin{aligned}
a : V \times V \to \mathbb{R} \qquad & a(\boldsymbol{w}, \boldsymbol{v}) := \nu \int_\Omega \nabla \boldsymbol{w} \cdot \nabla \boldsymbol{v} d\boldsymbol{x}, \\
b : V \times Q \to \mathbb{R} \qquad & b(\boldsymbol{v}, q) := -\int_\Omega q \nabla \cdot \boldsymbol{v} d\boldsymbol{x}, \\
c : V \times V \times V \to \mathbb{R} \quad & c(\boldsymbol{w}; \boldsymbol{z}, \boldsymbol{v}) := \int_\Omega [(\boldsymbol{w} \cdot \nabla)\boldsymbol{z}] \cdot \boldsymbol{v} = \sum_{i,j=1}^{d} \left(w_j \frac{\partial z_i}{\partial x_j}, v_i \right), \\
& (\boldsymbol{f}, \boldsymbol{v}) = \int_\Omega \boldsymbol{f} \boldsymbol{v} d\boldsymbol{x}.
\end{aligned}$$

Für die Geschwindigkeit aus dem Raum V_{div} ist die Kontinuitätsgleichung automatisch erfüllt und die Aufgabe (3.19) reduziert sich auf die Impulsgleichung

$$a(\boldsymbol{u}, \boldsymbol{v}) + c(\boldsymbol{u}; \boldsymbol{u}, \boldsymbol{v}) = (\boldsymbol{f}, \boldsymbol{v}) \quad \forall \boldsymbol{v} \in V_{\text{div}}. \tag{3.20}$$

Wenn $(\boldsymbol{u}, p)$ Lösung von (3.19) ist, dann ist $\boldsymbol{u}$ Lösung von (3.20) und umgekehrt. Diese Lösung existiert immer unter der Bedingung $\boldsymbol{f} \in H_{\text{div}}$, ist aber nicht unbedingt eindeutig. Sie kann z.B. nicht eindeutig sein, wenn die kinematische Viskosität ν klein bezüglich $\boldsymbol{f}$ ist (Temam [160], Kap. II, Abschnitt 4), z.B. für das Taylor-Problem von Strömungen zwischen unendlichen rotierenden Zylindern.

Die Zeitdiskretisierung und die Linearisierung der Navier-Stokes-Gleichungen (Abschnitt 3.1.1) führen zum Stokes-Problem, dessen Variationsformulierung sich wie folgt schreiben lässt:

$$\begin{cases}
\boldsymbol{u} \in V,\ p \in Q \text{ zu finden, dass} & \\
a(\boldsymbol{u}, \boldsymbol{v}) + b(\boldsymbol{v}, p) = (\boldsymbol{f}, \boldsymbol{v}) & \forall \boldsymbol{v} \in V \\
b(\boldsymbol{u}, q) = 0 & \forall q \in Q.
\end{cases} \tag{3.21}$$

3.4.2 Galerkin-Approximation

Das Gebiet Ω wird, wie im Abschnitt 3.2, mit Hilfe eines Dreieckgitters $\mathcal{T}_h$ oder Viereckgitters $\mathcal{Q}_h$ diskretisiert. Mit K wird ein Dreieck- oder Viereckelement des Gitters bezeichnet. Wir führen nun die von h abhängende endlich dimensionalen Räume $V_h \subset V$ und $Q_h \subset Q$ ein. Dann wird das Stokes-Problem (3.21) durch das folgende diskrete Problem approximiert:

$$\begin{cases}
\boldsymbol{u}_h \in V_h,\ p_h \in Q_h \text{ zu finden, dass} & \\
a(\boldsymbol{u}_h, \boldsymbol{v}_h) + b(\boldsymbol{v}_h, p_h) = (\boldsymbol{f}, \boldsymbol{v}_h) & \forall \boldsymbol{v}_h \in V_h \\
b(\boldsymbol{u}_h, q_h) = 0 & \forall q_h \in Q_h.
\end{cases} \tag{3.22}$$

Seien $\{\boldsymbol{\varphi}_i \,|\, i = 1, ..., N_u\}$ und $\{\psi_i \,|\, i = 1, ..., N_p\}$ die Basen der Räume V_h und Q_h entsprechend, werden die Näherungen $\boldsymbol{u}_h$ und p_h durch die folgenden Entwicklungen dargestellt:

$$\boldsymbol{u}_h(\boldsymbol{x}) = \sum_{i=1}^{N_u} u_i \boldsymbol{\varphi}_i(\boldsymbol{x}), \qquad p_h(\boldsymbol{x}) = \sum_{i=1}^{N_p} p_i \psi_i(\boldsymbol{x}). \tag{3.23}$$

Dann erhält man das lineare algebraische Problem

$$\begin{cases} F\hat{\boldsymbol{u}} + B^T\hat{\boldsymbol{p}} = \hat{\boldsymbol{f}} \\ B\hat{\boldsymbol{u}} = \mathbf{0} \end{cases} \tag{3.24}$$

wobei die Matrix- und die Vektorelemente durch die Basisfunktionen

$$F_{ij} = a(\boldsymbol{\varphi}_i, \boldsymbol{\varphi}_j), \qquad B_{ij} = b(\boldsymbol{\varphi}_i, \psi_j), \qquad \hat{f}_i = (\boldsymbol{f}_i, \boldsymbol{\varphi}_i) \tag{3.25}$$

bestimmt werden. Die Lösung des linearen Gleichungssystems (3.24) wird in den Abschnitten (5.2.5,5.2.6) betrachtet.

3.4.3 Stabilitätsbedingungen

Die Basisfunktionen werden normalerweise als Polynomfunktionen gewählt, dann werden die Finite-Elemente-Räume V_h und Q_h genauer durch

$$V_h = \left\{ \boldsymbol{\varphi} \in (H_0^1(\Omega))^d \mid \boldsymbol{\varphi}_{|K} \in P_k \quad \forall K \in \mathcal{T}_h \right\},$$

$$Q_h = \left\{ \psi \in L_0^2(\Omega) \mid \psi_{|K} \in P_k \quad \forall K \in \mathcal{T}_h \right\}$$

für ein Dreieckgitter und durch

$$V_h = \left\{ \boldsymbol{\varphi} \in (H_0^1(\Omega))^d \mid \boldsymbol{\varphi}_{|K} \in Q_k \quad \forall K \in \mathcal{Q}_h \right\},$$

$$Q_h = \left\{ \psi \in L_0^2(\Omega) \mid \psi_{|K} \in Q_k \quad \forall K \in \mathcal{Q}_h \right\}$$

für ein Viereckgitter definiert. Die Menge $P_k,\ k \geq 0$ ($Q_k,\ k \geq 0$) bezeichnet Polynome der Ordnung nicht größer als k bezüglich aller Variablen (jeder Variable) $x_1, ..., x_d$. Damit das diskrete Problem (3.22) eine stabile Approximation zum stetigen Problem (3.21) ist, wenn $h \to 0$, ist es entscheidend, dass der Raum $V_h \times Q_h$ die Kompatibilitätsbedingung (die sogenannte inf-sup- oder Ladyzhenskaya-Babuška-Brezzi-Bedingung)

$$\inf_{\phi \in Q_h} \left\{ \sup_{\boldsymbol{w} \in V_h} \frac{b(\boldsymbol{w}, \phi)}{\|\nabla \boldsymbol{w}\| \|\phi\|} \right\} \geq \gamma > 0 \tag{3.26}$$

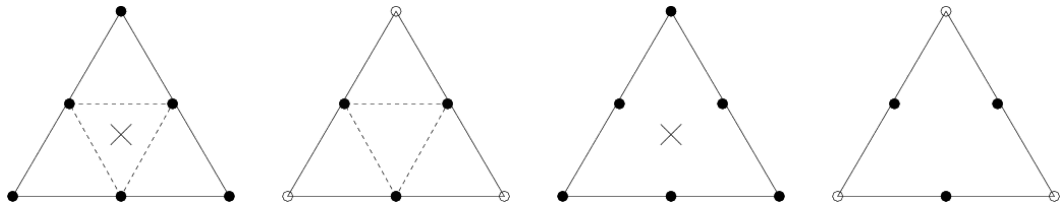
erfüllt. Dabei ist γ eine von h unabhängige Konstante.

Man muss nun die stabilen Paare der Ansatzfunktionen φ/ψ auswählen. Die einfachste Kombination ist stückweise lineare Geschwindigkeit / stückweise konstanter Druck auf einem Dreieckgitter (P_1/P_0-Element), die gerade ein Gegenbeispiel darstellt. Sie erfüllt die Kompatibilitätsbedingung (3.26) nicht und kann zum sogenannten Locking-Phänomen führen, wenn die Gleichung (3.22_2) nur die triviale Lösung $\boldsymbol{u}_h = \mathbf{0}$ zulässt. Genauso ist das Q_1/Q_0-Element instabil [68, 137], obwohl es immer noch breit angewendet wird. Es gibt zwei Strategien, die stabilen konformen Stokes-Elemente zu bekommen:

1. Man definiert das Geschwindigkeitsfeld auf einem verfeinerten Gitter (Abb. 3.5,a,b). Dabei muss der Druck auf dem gröberen Gitter mit der gleichen oder einer kleineren Approximationsordnung definiert werden, dann ist das Schema stabil [75].

2. Es wird eine höhere Approximationsordnung der Geschwindigkeit bezüglich des Drucks benutzt. Die Elemente P_2/P_0 (Abb. 3.5,c) und Q_2/Q_0 sind stabil und zeigen lineare Konvergenz. Das verallgemeinerte Taylor-Hood-Element P_m/P_{m-1} ($m \geq 2$) (Abb. 3.5,d) ist auch stabil [21, 22, 25].

Es gibt auch die Möglichkeit, Stabilisierungsmethoden auf instabile konforme Elemente anzuwenden, wie z.B. auf das divergenzfreien Q_1/Q_1-Element [140].



a) P_1^M/P_0 (Makro) b) P_1^M/P_1 (Makro) c) P_2/P_0 d) P_2/P_1

Abbildung 3.5: Beispiele der Dreieck-Stokes-Elemente. ∘ Geschwindigkeit- und Druck-Freiheitsgrad; • Geschwindigkeit-Freiheitsgrad; × Druck-Freiheitsgrad.

Eine andere Klasse der stabilen Elemente finden sich in der nichtkonformen Approximation. Das entspricht dem Fall $V_h \not\subset V$. Die Galerkin-Methode (3.22) muss nun durch die verallgemeinerte Galerkin-Methode

$$\begin{cases} \boldsymbol{u}_h \in V_h,\ p_h \in Q_h \text{ zu finden, dass} & \\ a_h(\boldsymbol{u}_h, \boldsymbol{v}_h) + b_h(\boldsymbol{v}_h, p_h) = (\boldsymbol{f}, \boldsymbol{v}_h) & \forall \boldsymbol{v}_h \in V_h \\ b_h(\boldsymbol{u}_h, q_h) = 0 & \forall q_h \in Q_h, \end{cases} \tag{3.27}$$

ersetzt werden, wobei die diskreten Bilinearformen $a_h(\cdot\,,\cdot)$ und $b_h(\cdot\,,\cdot)$ nun zellenweise integriert werden müssen.

3.5 Äquivalente FV/FE-Formulierungen

Die Galerkin-FEM (GFEM) ist dafür bekannt, die optimale Approximation für selbst-adjungierte Probleme darzustellen, obwohl für andere und insbesondere Erhaltungsgesetze die Frage der Wahl zwischen FVM und FEM immer noch offen ist und in der Literatur aktiv diskutiert wird [70, 84, 189]. Die FVM ist mehr bei Ingenieuren beliebt und wird ohne ausreichende mathematische Analyse von Konvergenz und Stabilität benutzt [86, 87, 178], da sie die lokale Konservativität ausdrückt. Dafür werden aber die Randintegrale der Kontrollvolumina durch Approximationsformel niedriger Ordnung (normalerweise erster oder zweiter Ordnung) errechnet. Die FEM ist dagegen sehr gut theoretisch unterstützt [94, 160], wobei Ansatzfunktionen beliebiger Ordnung (p-Version) zugelassen sind. Die beiden Methoden stammen allerdings aus der Methode der gewichteten Residuen [188]. Deshalb meint man auch, dass die FVM ein partikulärer Fall von nicht-Galerkinschen FEM ist. Ein gutes Beispiel dafür ist die Kovolumen-Methode [32, 33, 34], die als eine Petrov-Galerkinsche FEM dargestellt werden kann [187]. Der Zusammenhang mit einer Petrov-Galerkin-FEM wurde auch von Li [98] benutzt, um die Konvergenz und Stabilität von FVM für die Advektions-Diffusions-Gleichung nachzuweisen. Die duale gemischte FEM wurde als

eine FVM von Baranger [1, 12] betrachtet. Die elliptischen Probleme kommen noch häufig in diesem Kontext vor [78, 128, 167, 168], da sie repräsentativ sind und keine besonderen Voraussetzungen auf die Wahl der Ansatzfunktionen anfordern.

Das diskrete Stokes-Problem (3.22) benötigt aber kompliziertere stabile Stokes-Elemente (z.B. Abb. 3.5,a,b,c) wegen der Geschwindigkeit-Druck-Kopplung. Die exakte Integration mit diesen stabilen Elementen und Umformulierung lässt sich als eine FVM auf einem dualen Gitter darstellen und damit die Konservativität nachweisen. Diese Umformulierung der GFEM in die FVM wird hier mit Hilfe von bekannten stabilen Stokes-Elementen niedriger Ordnung durchgeführt.

3.5.1 Poisson-Problem

Das klassische Beispiel der Äquivalenz zwischen den FE- und FV-Formulierungen nutzt einen Übergang von einer Delaunay-Triangulierung zum dualen Voronoi-Diagramm für die Galerkin-Formulierung mit linearen Ansatzfunktionen für das Poisson-Problem ([91] S.308 und [167]). Der Hauptschwierigkeit bei diesem Vergleich besteht darin, dass die Ansatzfunktionen üblicherweise bezüglich Koordinaten der Gitterknoten ausgedrückt werden. Die Randintegrale in der FV-Formulierung werden dagegen durch Gitterkanten aufgeschrieben. Um dieses Problem zu lösen, bilden wir einen Zusammenhang zwischen einem lokalen Koordinatensystem und Geometriegrößen des Gitters.

$b = |A_1 H|$

$a = |A_2 A_3|$

$c = \cot \theta_{12}$

$A_1(a - bc, b)$

$A_2(0, 0)$

$A_3(a, 0)$

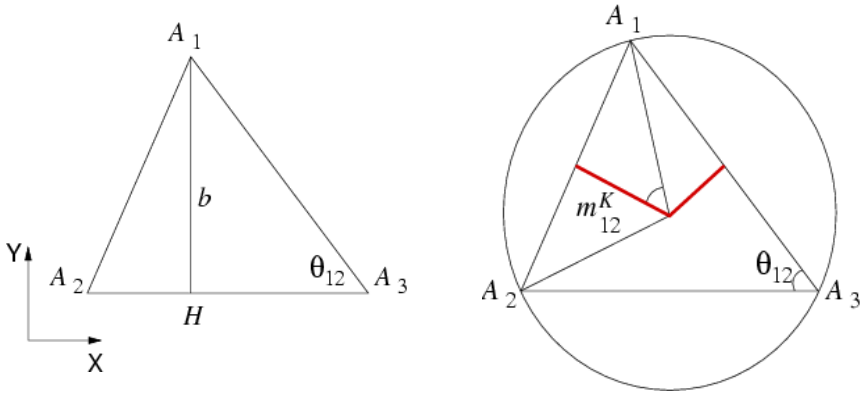


Abbildung 3.6: Das gewählte Koordinatensystem und die Basis eines Dreiecks

Wir betrachten ein allgemeines Dreieck $A_1A_2A_3$ mit den Ecken $A_1(x_1, y_1)$, $A_2(x_2, y_2)$, $A_3(x_3, y_3)$ und führen eine Basis ein (im Unterschied zu den Beweisen in [91], S.308, und [167]), durch die das Dreieck eindeutig definiert wird. Diese Basis wird auch bei der Behandlung von komplizierteren Stokes-Elementen gebraucht, insbesondere wenn die Gestalt der Ansatzfunktionen und deren Ableitungen nicht offensichtlich sind. Sei die x-Achse des Koordinatensystems mit dem Ursprung im Punkt A_2 parallel zur Seite A_2A_3 des Dreiecks. Dann ist das Dreieck $A_1A_2A_3$ eindeutig durch die Länge a der Seite A_2A_3, die zu dieser Kante senkrecht stehende Höhe b und den gegenüber zur Kante A_1A_2 liegenden Winkel θ_{12} bestimmt (Abb. 3.6). Als die Messung des Winkels θ_{12} wird der Kotangens $\cot \theta_{12}$ genommen, weil er einen direkten Übergang vom Dreieck zum entsprechenden dualen Voronoi-Polygon zulässt, d.h.

$$c = \cot \theta_{12} = \frac{2m_{12}^K}{a_{12}}, \tag{3.28}$$

wobei $a_{12} = |A_1A_2|$ und m_{12}^K der Abstand des Umkreiszentrums des Dreiecks K zum Mittelpunkt der Kante A_1A_2 ist. Wenn zwei Dreiecken K und $\tilde{K}$ eine gemeinsame Kante A_1A_2 besitzen, bezeichnen wir auch die Länge der Voronoi-Kante mit

$$m_{12} = m_{12}^K + m_{12}^{\tilde{K}}. \tag{3.29}$$

Die zum Dreieck K gehörenden Ansatzfunktionen φ_i^K $(i = 1, 2, 3)$ können nun durch die Größen a, b und c mit Hilfe der neuen Koordinaten der Ecken $A_1(a - bc, b)$, $A_2(0, 0)$ und $A_3(a, 0)$ ausgedrückt werden, was im nächsten Satz angewendet wird.

Satz 3.1. *Seien $\mathcal{T}_h$ eine Delaunay-Triangulierung von Ω und $NK(k)$ die Menge aller Knoten, die benachbarten zum Knoten A_k sind. Dann stimmt die GFEM mit $\varphi_i \in P_1$ für das Poisson-Problem auf der Triangulierung $\mathcal{T}_h$ vollständig mit der FVM für dasselbe Problem auf dem dualen Voronoi-Diagramm $\mathcal{V}_h$ überein mit der folgenden Randintegral-Approximation*

$$\int_{\partial V_k} \nabla u \cdot \tilde{\boldsymbol{n}} ds \approx \sum_{i \in NK(k)} \frac{u_i - u_k}{a_{ik}} m_{ik} = - \sum_{i \in \{k, NK(k)\}} \int_\Omega \nabla\varphi_i \nabla\varphi_k dx\, u_i, \tag{3.30}$$

wobei $\tilde{\boldsymbol{n}}$ die äußere Einheitsnormale zum Rand ∂V_k des Voronoi-Polygons V_k ist.

Beweis: Die erste Approximation in (3.30) ist nichts anderes als die Differenzen-Annäherung zweiter Ordnung für die Geschwindigkeit u im Mittelpunkt der Kante A_iA_k, multipliziert mit der Länge der Voronoi-Kante so, dass die ganze Beziehung die Integral-Approximation darstellt. Deshalb muss man nur die zweite Gleichheit bzw. den Übergang von FE-Formulierung zur Approximation auf der Voronoi-Kante beweisen.

Wir betrachten zuerst das Skalarprodukt $\nabla\varphi_i\nabla\varphi_k$ nur in einem Dreieck K, wo dieselbe Basis wie in der Abbildung 3.6 definiert ist. Die linearen Ansatzfunktionen φ_i^K sehen dann wie folgt aus:

$$\begin{aligned}
\varphi_1^K &= \tfrac{1}{b} y, \\
\varphi_2^K &= -\tfrac{1}{a} x - \tfrac{c}{a} y + 1, \\
\varphi_3^K &= \tfrac{1}{a} x + \left(\tfrac{c}{a} - \tfrac{1}{b}\right) y + 1.
\end{aligned}$$

Für $k = 1$ gilt

$$\sum_{i=1}^{3} \int_K \nabla\varphi_i \nabla\varphi_1 d\boldsymbol{x}\, u_i = \frac{a}{2b} u_1 - \frac{c}{2} u_2 - \frac{a - bc}{2b} u_3 \tag{3.31}$$

Aus elementaren geometrischen Beziehungen im Dreieck ergibt sich

$$\frac{a}{2b} = \frac{1}{2} \cot\theta_{12} + \frac{1}{2} \cot\theta_{13}, \tag{3.32}$$

$$\frac{a - bc}{2b} = \frac{1}{2} \cot\theta_{13}. \tag{3.33}$$

Die letzten zwei Gleichungen zusammen mit (3.28) gestatten die Entwicklung von (3.31) zu

$$\sum_{i=1}^{3} \int_K \nabla\varphi_i \nabla\varphi_1 d\boldsymbol{x}\, u_i = \frac{m_{12}^K}{a_{12}}(u_1 - u_2) + \frac{m_{13}^K}{a_{13}}(u_1 - u_3). \tag{3.34}$$

Man kann leicht nachprüfen, dass die Integrale für $k = 2$ und $k = 3$ genau so aussehen, d.h. die Beziehung (3.34) von der Auswahl des Koordinatensystems unabhängig ist. Die Summe über alle Dreiecke, die den Knoten A_k gemeinsam haben, und Berücksichtigung von (3.29) liefern unmittelbar die Behauptung.

□

Zum vollständigen Vergleich der FEM mit $\varphi \in P_1$ und der FVM für das Poisson-Problem lassen sich noch die Approximationen der rechten Seite betrachten.

Bemerkung 3.1. *Wir bezeichnen $\hat{f}_G$ und $\hat{f}_V$ als die GFEM-Approximation auf $\mathcal{T}_h$ und die FVM-Approximation auf einem dualen Gitter. Dabei wird das duale Gitter durch die Verbindung von Seiten-Mittelpunkten mit einem inneren (nicht unbedingt Voronoi-) Punkt gebildet. Dann ist die duale Norm (Hackbusch [78], Lemma 3.2.2):*

$$|\hat{f}_G - \hat{f}_V|_{-1} \leq \frac{h}{\sqrt{2}} \|f\|_0. \tag{3.35}$$

3.5.2 Stokes-Gleichungen

Die eingeführten geometrischen Bezeichnungen und die erwähnten Eigenschaften werden weiter im Kontext der LBB-stabilen Galerkin-Methoden für die Stokes-Gleichungen komponentenweise in zwei Dimensionen

$$\begin{cases} \nu \sum_{i=1}^{N_u} \int_\Omega \nabla\varphi_i \nabla\varphi_k \, dxdy \, u_i - \sum_{j=1}^{N_p} \int_\Omega \psi_j \dfrac{\partial\varphi_k}{\partial x} \, dxdy \, p_j = \int_\Omega f_1 \varphi_k \, dxdy \\ \nu \sum_{i=1}^{N_u} \int_\Omega \nabla\varphi_i \nabla\varphi_k \, dxdy \, v_i - \sum_{j=1}^{N_p} \int_\Omega \psi_j \dfrac{\partial\varphi_k}{\partial y} \, dxdy \, p_j = \int_\Omega f_2 \varphi_k \, dxdy \\ \sum_{i=1}^{N_u} \int_\Omega \psi_l \left(\dfrac{\partial\varphi_i}{\partial x} u_i + \dfrac{\partial\varphi_i}{\partial y} v_i \right) dxdy = 0 \qquad k = 1, ..., N_u \qquad l = 1, ..., N_p \end{cases} \tag{3.36}$$

betrachtet, wobei $\boldsymbol{u} = (u, v)^T$ und $\boldsymbol{f} = (f_1, f_2)^T$ sind. Die zweidimensionale FV-Formulierung mit denselben Bezeichnungen und zusätzlich der Einheitsnormale $\boldsymbol{n} = (n^x, n^y)^T$ lautet dann:

$$\begin{cases} -\nu \int_{\partial\Omega_k} \nabla u \cdot \boldsymbol{n} do + \int_{\partial\Omega_k} p n^x do = \int_{\Omega_k} f_1 dxdy \\ -\nu \int_{\partial\Omega_k} \nabla v \cdot \boldsymbol{n} do + \int_{\partial\Omega_k} p n^y do = \int_{\Omega_k} f_2 dxdy \qquad k = 1, ..., N_u \\ \int_{\partial\tilde{\Omega}_l} (u n^x + v n^y) \, do = 0 \qquad\qquad l = 1, ..., N_p \end{cases} \tag{3.37}$$

Die Integrale über das Gebiet Ω in (3.36) unterscheiden sich von Null nur in Teilgebieten, wo $\varphi_k \neq 0$ ist. Auf gleichen Teilgebieten müssen auch die Volumina Ω_k

in (3.37) definiert werden. In der Kontinuitätsgleichung (3.36_3) beschränkt sich das Integral über Ω auf ein anderes Teilgebiet, wobei $\psi_l \neq 0$ gilt. Deshalb muss die FVM für die Kontinuitätsgleichung (3.37_3) auf einem anderen Volumen $\tilde{\Omega}_l$ formuliert werden, da φ_k und ψ_l für die stabile Stokes-Elemente unterschiedlich sind.

P_1^M/P_0-Stokes-Element

Die Triangulierung $\mathcal{T}_h$ des Gebietes Ω wird durch Verbindung von Kanten-Mittelpunkten verfeinert. So bildet sich die neue Triangulierung $\mathcal{T}_{h/2}$ mit dem dualen Voronoi-Diagramm $\mathcal{V}_{h/2}$. Das LBB-stabile und häufig benutzte Element P_1^M/P_0 besteht aus den stückweise linearen Ansatzfunktionen auf der Triangulierung $\mathcal{T}_{h/2}$ für die Geschwindigkeit und dem auf jedem Dreieck der Triangulierung $\mathcal{T}_h$ konstanten Druck (Abb. 3.5,a). Dieses Element zusammen mit der nachfolgenden Modifikation P_1^M/P_1 besitzt eine lineare Konvergenzrate und wurde auch das beste Element erster Ordnung von Gunzburger [75] genannt.

Wir gehen von der Galerkin-Formulierung (3.36) aus und wandeln die diskreten Gleichungen so um, dass eine FV-Formulierung für dasselbe Problem möglicherweise auf einem dualen Gitter entsteht. Die entsprechenden Gleichungssystem-Matrizen stimmen dann vollständig überein.

$A_1(a-bc, b)$ $\quad A_{12}(\frac{a-bc}{2}, \frac{b}{2})$

$A_2(0,0)$ $\quad A_{13}(a-\frac{bc}{2}, \frac{b}{2})$

$A_3(a,0)$ $\quad A_{23}(\frac{a}{2}, 0)$

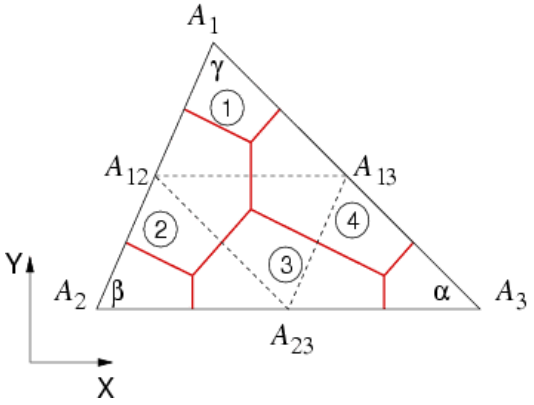


Abbildung 3.7: Ein verfeinertes Dreieck mit dem dualen Voronoi-Diagramm

Wir nummerieren o.B.d.A. die Ecken eines allgemeinen Dreiecks mit A_1, A_2, A_3 und bezeichnen den Mittelpunkt der Kante A_iA_j durch A_{ij} $(i,j=1,2,3,\ i\neq j)$ (Abb. 3.7). Für die entstandenen vier gleichen Dreiecken $K_1 = \triangle A_1A_{12}A_{13}$, $K_2 = \triangle A_2A_{12}A_{23}$, $K_3 = \triangle A_{12}A_{13}A_{23}$ und $K_4 = \triangle A_3A_{13}A_{23}$ werden die charakteristischen Größen $\{a,b,c\}$ und das Koordinatensystem, wie in der Abbildung 3.6, benutzt. Dadurch kann man die Hut-Funktionen φ_j^i des Knotens A_j im Dreieck K_i genau definieren (A.1). Mit Hilfe der Bezeichnung $\alpha = \theta_{12}$, $\beta = \theta_{13}$ und $\gamma = \theta_{23}$ schreiben wir außer (3.32) und (3.33) noch weitere geometrische Dreieck-Eigenschaften auf, die einfach zu prüfen sind:

$$m_{12}^K \cos\beta = m_{13}^K \cos\alpha, \tag{3.38}$$

$$m_{12}^K \sin\beta + m_{13}^K \sin\alpha = \frac{a}{2}, \tag{3.39}$$

$$m_{12}^K \cos\beta + m_{23}^K = m_{13}^K \cos\alpha + m_{23}^K = \frac{b}{2}, \tag{3.40}$$

$$m_{12}^K = 2m_{1,12}^K = 2m_{2,12}^K, \qquad m_{12}^K = 2m_{13,23}^K = m_{13,23}. \tag{3.41}$$

Außerdem gilt

$$\tilde{\boldsymbol{n}}_{12,1} = (\cos\beta, \sin\beta)^T, \qquad \tilde{\boldsymbol{n}}_{13,1} = (\cos\alpha, \sin\alpha)^T, \qquad \tilde{\boldsymbol{n}}_{12,13} = (1,0)^T, \tag{3.42}$$

$$\boldsymbol{n}_{12} = (-\sin\beta, \cos\beta)^T, \qquad \boldsymbol{n}_{13} = (\sin\alpha, \cos\alpha)^T, \qquad \boldsymbol{n}_{23} = (0,-1)^T \tag{3.43}$$

für die äußere Einheitsnormale $\boldsymbol{n}_{ij} = (n_{ij}^x, n_{ij}^y)^T$ zur Kante A_iA_j und $\tilde{\boldsymbol{n}}_{ij} = (\tilde{n}_{ij}^x, \tilde{n}_{ij}^y)^T$ der Einheitsvektor zur Voronoi-Kante m_{ij} in die Richtung von i nach j.

Der Satz 3.1 gilt für den Diffusionsterm der Gleichung (3.36) auf jedem Dreieck K_i, $i = 1,..,4$, da sie ähnlich zum verfeinerten Dreieck sind und das modifizierte Gitter auch ein Delaunay-Gitter ist.

Auf dem ganzen Dreieck $A_1A_2A_3$ gibt es nur eine Ansatzfunktion für den Druck ψ^K, die überall im Dreieck den Wert 1 annimmt und auf dem Rand und dem Rest des Gebietes Ω verschwindet. Dann liefert die exakte Integration der Druckterme in (3.36) mit Berücksichtigung von (3.38)-(3.42), z.B. für $k = 1$:

$$\begin{cases} \displaystyle\int_K \frac{\partial\varphi_1}{\partial x}\,dxdy\,p^K = 0\,p^K & = (-m_{1,12}^K\cos\beta + m_{1,13}^K\cos\alpha)\,p^K \\ & = (m_{1,12}^K\tilde{n}_{1,12}^x + m_{1,13}^K\tilde{n}_{1,13}^x)\,p^K \\ \displaystyle\int_K \frac{\partial\varphi_1}{\partial y}\,dxdy\,p^K = \frac{a}{4}\,p^K & = (m_{1,12}^K\sin\beta + m_{1,13}^K\sin\alpha)\,p^K \\ & = -(m_{1,12}^K\tilde{n}_{1,12}^y + m_{1,13}^K\tilde{n}_{1,13}^y)\,p^K \end{cases} \tag{3.44}$$

und für den Mittelpunkt A_{12}

$$\begin{cases} \displaystyle\int_K \frac{\partial\varphi_{12}}{\partial x}\,dxdy\,p^K = -\frac{b}{2}p^K = & -(m_{12,23}\cos\alpha + m_{12,13})\,p^K \\ = & -(m_{12,23}\tilde{n}_{12,23}^x + m_{12,13}\tilde{n}_{12,13}^x \\ & +m_{12,1}^K\tilde{n}_{12,1}^x + m_{12,2}^K\tilde{n}_{12,2}^x)\,p^K, \\ \displaystyle\int_K \frac{\partial\varphi_{12}}{\partial y}\,dxdy\,p^K = \frac{a-bc}{2}p^K = & (m_{12,23}\sin\alpha)\,p^K \\ = & (m_{12,23}\tilde{n}_{12,23}^y + m_{12,13}\tilde{n}_{12,13}^y \\ & +m_{12,1}^K\tilde{n}_{12,1}^y + m_{12,2}^K\tilde{n}_{12,2}^y)\,p^K. \end{cases} \tag{3.45}$$

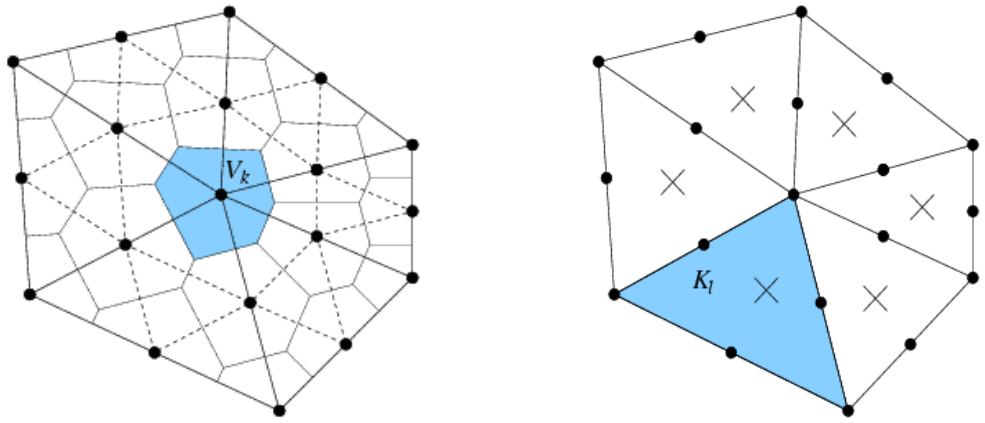
In den rechten Seiten der letzten Gleichungen wurden die Glieder addiert, deren Summe gleich Null ist, so dass sich alle in K liegenden Voronoi-Kanten, multipliziert mit der entsprechenden Projektion der Einheitsnormale, ergeben. Man kann sich allerdings mit Hilfe derselben Basis $\{a, b, c\}$ leicht überzeugen, dass die gleichen Beziehungen auch für die anderen φ_i gelten und die Addition von verschwindenden Gliedern sinnvoll ist. Vollständige Formeln (A.2)-(A.5) sind im Anhang angegeben. Damit ist das Resultat unabhängig von der Auswahl des Koordinatensystems.

Wir führen nun die Menge $I = \{1, 2, 3, 12, 13, 23\}$ aller Indizes der Geschwindigkeits-Freiheitsgrade des Dreiecks K ein und integrieren die Galerkin-Formulierung

für die Kontinuitätsgleichung (3.36_3) über das Dreieck K

$$\begin{aligned}\sum_{i\in I}\int_K\left(\frac{\partial\varphi_i}{\partial x}u_i+\frac{\partial\varphi_i}{\partial y}v_i\right)dxdy = \;& 0u_1-\tfrac{b}{4}u_2+\tfrac{b}{4}u_3-\tfrac{b}{2}u_{12}+\tfrac{b}{2}u_{13}+0u_{23}\\ &+\tfrac{a}{4}v_1-\tfrac{bc}{4}v_2-\tfrac{a-bc}{4}v_3+\tfrac{a-bc}{2}v_{12}+\tfrac{bc}{2}v_{13}-\tfrac{a}{2}v_{23}\\ =\;& -\bar{u}_{12}a_{12}\sin\beta+\bar{u}_{13}a_{13}\sin\alpha\\ &+\bar{v}_{12}a_{12}\cos\beta+\bar{v}_{13}a_{13}\cos\alpha-\bar{v}_{23}a,\end{aligned} \tag{3.46}$$

wobei $\bar{u}_{ij}=\frac{u_i+2u_{ij}+u_j}{4}$ und $\bar{v}_{ij}=\frac{v_i+2v_{ij}+v_j}{4}$ $(i,j=1,2,3,\ i\neq j)$, was mit Berücksichtigung von (3.43) eine konservative FV-Approximation auf dem Dreieck K darstellt.



a) für die Impulsgleichungen b) für die Kontinuitätsgleichung

Abbildung 3.8: Finite Volumen des P_1^M/P_0-Stokes-Elementes für die Impulsgleichung und die Kontinuitätsgleichung. • Geschwindigkeit-Freiheitsgrad; × Druck-Freiheitsgrad.

Seien $NK(k)$ bzw. $NT(k)$ ($NK'(k)$ bzw. $NT'(k)$) die Mengen aller zum A_k benachbarten Knoten bzw. Dreiecken der Triangulierung $\mathcal{T}_h$ ($\mathcal{T}_{h/2}$). Die Summe über alle Geschwindigkeits- bzw. Druck-Freiheitsgrade des Gitters liefert die zum P_1^M/P_0-Element äquivalente FVM (3.47), die eine Node-Centered-FVM für die Impulsgleichungen und eine Cell-Centered-FVM für die Kontinuitätsgleichung darstellen:

$$\begin{cases} -\displaystyle\int_{\partial V_k}\nabla\boldsymbol{u}\cdot\tilde{\boldsymbol{n}}do+\int_{\partial V_k}p\tilde{\boldsymbol{n}}do=-\sum_{i\in NK'(k)}\frac{\boldsymbol{u}_i-\boldsymbol{u}_k}{a_{ik}}m_{ik}+\sum_{j\in NT'(k)}p_j\tilde{\boldsymbol{n}}\,|\partial V_k^{K_j}|+O(h)\\ \displaystyle\int_{\partial K_l}\boldsymbol{u}\cdot\boldsymbol{n}do=\sum_{A_i,A_j\in K_l}\left(\frac{\boldsymbol{u}_i+2\boldsymbol{u}_{ij}+\boldsymbol{u}_j}{4}\boldsymbol{n}_{ij}\right)a_{ij}+O(h)\end{cases} \tag{3.47}$$

wobei (Abb. 3.8,a,b), $k=1,...,N_u$, $l=1,...,N_p$ und u_{ij} für die Geschwindigkeits-Approximation im Mittelpunkt der Kante A_iA_j steht. Die Finite-Volumina V_k und K_l sind in der Abbildung (3.8,a,b) entsprechend für die Impulsgleichungen und die Kontinuitätsgleichung repräsentiert. Als $|\partial V_k^{K_j}|$ wird die Länge des im Dreieck K_j liegenden Voronoi-Polygon-Randes ∂V_k bezeichnet. Die äußere Einheitsnormale $\boldsymbol{n}=(n^x,n^y)^T$ eines Gitterdreiecks unterscheidet sich von der Einheitsnormale $\tilde{\boldsymbol{n}}=(\tilde{n}^x,\tilde{n}^y)^T$ eines Voronoi-Polygons.

P_1^M/P_1-Stokes-Element

Diese stabile und linear konvergierende Modifikation des P_1^M/P_0-Elementes besitzt drei Freiheitsgrade für den Druck in den Dreiecks-Ecken (Abb. 3.5,b). Da für eine konforme Triangulierung $N_T = 2N_K - N_R - 2$ gilt, wobei N_T, N_K und N_R die Anzahl von Dreieckelementen, Knoten und Randknoten entsprechend bezeichnen (Abschnitt 3.2), enthält das P_1^M/P_1-Element auf dem ganzen Gitter normalerweise weniger Druckpunkte. Mit den auf dem Gitter $\mathcal{T}_{h/2}$ linearen Ansatzfunktionen (A.1) für die Geschwindigkeit und den auf dem Gitter $\mathcal{T}_h$ linearen Ansatzfunktionen (A.6) für den Druck ändert sich nun die Integration der Druck-Terme im Vergleich zum Element P_1^M/P_0. Mit Hilfe der Eigenschaften (3.38),(3.39) und der Abbildung 3.7 erhält man für die Ansatzfunktion φ_1:

$$\begin{aligned}
\sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_1}{\partial x} dxdy\, p_j &= 0 \\
&= \tfrac{4p_1+p_2+p_3}{6}(-m_{1,12}^K \cos\beta + m_{1,13}^K \cos\alpha), \\
\sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_1}{\partial y} dxdy\, p_j &= \tfrac{a}{6}p_1 + \tfrac{a}{24}p_2 + \tfrac{a}{24}p_3 \\
&= \tfrac{4p_1+p_2+p_3}{6}(m_{1,12}^K \sin\beta + m_{1,13}^K \sin\alpha).
\end{aligned} \tag{3.48}$$

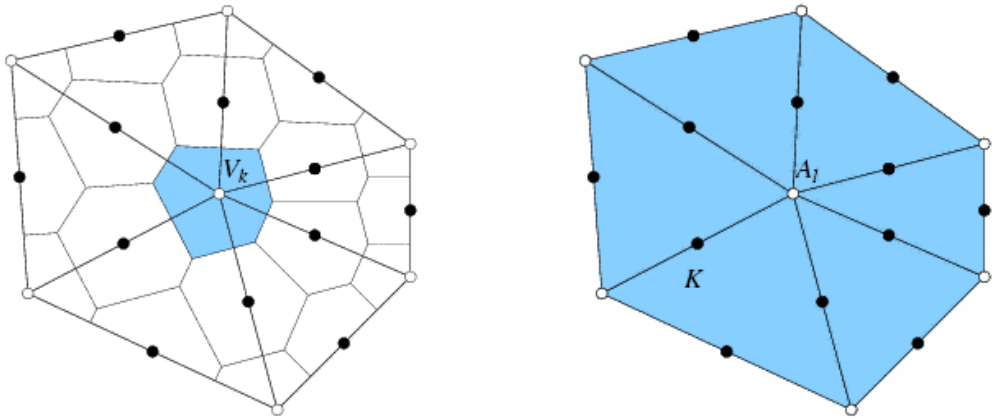
Die nächste Umformulierung für den Freiheitsgrad im Kantenmittelpunkt muss der letztgenannten Approximation (3.48) angepasst sein, deshalb gilt

$$\begin{aligned}
\sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_{12}}{\partial x} dxdy\, p_j &= -\tfrac{b}{4}p_1 - \tfrac{b}{8}p_2 - \tfrac{b}{8}p_3 \\
&= -\tfrac{4p_1+p_2+p_3}{6} m_{1,12}^K \cos\alpha - \tfrac{2p_1+p_2+p_3}{4} m_{12,13}^K \\
&\quad -\tfrac{p_1+2p_2+p_3}{4} m_{12,23}^K \cos\beta + \tfrac{p_1+4p_2+p_3}{6} m_{2,12}^K \cos\alpha, \\
\sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_{12}}{\partial y} dxdy\, p_j &= \tfrac{a-2bc}{8}p_1 + \tfrac{2a-bc}{8}p_2 + \tfrac{a-bc}{8}p_3 \\
&= -\tfrac{4p_1+p_2+p_3}{6} m_{1,12}^K \sin\alpha + \tfrac{p_1+2p_2+p_3}{4} m_{12,23}^K \sin\beta \\
&\quad +\tfrac{p_1+4p_2+p_3}{6} m_{2,12}^K \sin\alpha.
\end{aligned} \tag{3.49}$$

Die Summe solcher Integrale über den Dreiecken des Trägers der Funktion ψ_l kann man in allgemeiner Form folgendermaßen darstellen

$$\begin{cases}
\displaystyle\sum_{K=A_iA_jA_k\in\mathcal{T}_h} \bar{p}_{kij}\left(m_{k,ik}^K \tilde{\boldsymbol{n}}_{k,ik} + m_{k,jk}^K \tilde{\boldsymbol{n}}_{k,jk}\right) \approx \int_{\partial V_k} p\tilde{\boldsymbol{n}} do \qquad (A_k \notin \mathcal{T}_{h/2}) \\
\displaystyle\sum_{K=A_iA_jA_k\in\mathcal{T}_h} (\bar{p}_{kij} m_{k,ik}^K \tilde{\boldsymbol{n}}_{ik,k} + \bar{p}_{kij}^* m_{ik,jk} \tilde{\boldsymbol{n}}_{ik,jk} \\
\qquad +\bar{p}_{ijk} m_{ik,i}^K \tilde{\boldsymbol{n}}_{i,ik} + \bar{p}_{ijk}^* m_{ik,ij} \tilde{\boldsymbol{n}}_{ik,ij}) \approx \displaystyle\int_{\partial V_{ik}} p\tilde{\boldsymbol{n}} do \quad (A_{ik} \in \mathcal{T}_{h/2}),
\end{cases} \tag{3.50}$$

wobei $\bar{p}_{ijk} = \frac{4p_i+p_j+p_k}{6}$, $\bar{p}_{ijk}^* = \frac{2p_i+p_j+p_k}{4}$ und $k = 1, ..., N_u$, $l = 1, ..., N_p$. Die sämtlichen Integrationsergebnisse für alle Freiheitsgrade findet man im Anhang A.1.



a) für die Impulsgleichungen b) für die Kontinuitätsgleichung

Abbildung 3.9: Finite Volumen des P_1^M/P_1-Stokes-Elementes für die Impulsgleichung und die Kontinuitätsgleichung. • Geschwindigkeit-Freiheitsgrad; × Druck-Freiheitsgrad.

Die exakte Integration der GFEM für die Kontinuitätsgleichung, z.B. für ψ_1, ist

$$\sum_{i\in I}\int_K \psi_1\left(\frac{\partial\varphi_i}{\partial x}u_i+\frac{\partial\varphi_i}{\partial y}v_i\right)dxdy = 0u_1-\tfrac{b}{24}u_2+\tfrac{b}{24}u_3-\tfrac{b}{4}u_{12}+\tfrac{b}{4}u_{13}+0u_{23} \\ +\tfrac{a}{6}v_1-\tfrac{bc}{24}v_2-\tfrac{a-bc}{24}v_3+\tfrac{a-2bc}{8}v_{12}-\tfrac{a-2bc}{8}v_{13}-\tfrac{a}{8}v_{23}. \tag{3.51}$$

Diese Beziehung versuchen wir durch die Voronoi-Kanten m_{ij}^K mittels (3.39,3.40) auszudrücken. Das liefert aber keine gewünschte Approximation, weil die gegenüberliegende Voronoi-Kante m_{23}^K (m_{13}^K,m_{12}^K) aus dem Ausdruck für die Funktion ψ_1 (ψ_2,ψ_3) nicht eliminiert werden kann. Damit kann man keinen geschlossenen Voronoi-Polygon aus dem Träger der Funktion ψ_l ($l=1,2,3$) gewinnen. Eine andere Möglichkeit ist, die Beziehungen (3.51) durch die Dreieckseiten als die Summe

$$a_{12}\boldsymbol{n}_{12}\cdot\bar{\boldsymbol{u}}_{12}+a_{13}\boldsymbol{n}_{13}\cdot\bar{\boldsymbol{u}}_{13}+a_{23}\boldsymbol{n}_{23}\cdot\bar{\boldsymbol{u}}_{23}, \tag{3.52}$$

$$\bar{\boldsymbol{u}}_{ij}=C_1\boldsymbol{u}_1+C_2\boldsymbol{u}_2+C_3\boldsymbol{u}_3+C_4\boldsymbol{u}_{12}+C_5\boldsymbol{u}_{13}+C_6\boldsymbol{u}_{23} \tag{3.53}$$

zu beschreiben, wobei der Normalvektor $\boldsymbol{n}_{ij}$ in (3.43) und in Abbildung (3.7) definiert ist. Mit der Berücksichtigung von

$$a_{23}=a,\qquad a_{12}n_{12}^x=a_{13}n_{13}^x=b,\qquad a_{12}n_{12}^y=a-bc,\qquad a_{13}n_{13}^y=bc \tag{3.54}$$

kann man aus (3.51) leicht sehen, dass die Approximation (3.53) mit gleichen Koeffizienten C_i ($i=1,...,6$) für alle Seiten nicht erfüllt werden kann. Dann suchen wir für die zu A_l gegenüberliegende Seite die verschiedenen von C_i Koeffizienten. Wir wollen zusätzlich, dass die Approximation auf zu A_l benachbarten Seiten in der Summe über alle Dreiecke $K\in NT(l)$ (in inneren Seiten des Trägers ψ_l) gegenseitig verschwindet. Mit dieser Bedingung kann die Approximation $\bar{\boldsymbol{u}}_{ij}$ eindeutig bestimmt

werden, z.B. für ψ_1 gilt:

$$\begin{aligned}
\boldsymbol{u}_1 &= \tfrac{1}{6}a_{12}\boldsymbol{n}_{12} + \tfrac{1}{6}a_{13}\boldsymbol{n}_{13},\\
\boldsymbol{u}_2 &= \tfrac{1}{24}a_{12}\boldsymbol{n}_{12} + \tfrac{1}{24}a_{23}\boldsymbol{n}_{23},\\
\boldsymbol{u}_3 &= \tfrac{1}{24}a_{13}\boldsymbol{n}_{13} + \tfrac{1}{24}a_{23}\boldsymbol{n}_{23},\\
\boldsymbol{u}_{12} &= \tfrac{1}{6}a_{12}\boldsymbol{n}_{12} + \tfrac{1}{8}a_{23}\boldsymbol{n}_{23},\\
\boldsymbol{u}_{13} &= \tfrac{1}{6}a_{13}\boldsymbol{n}_{13} + \tfrac{1}{8}a_{23}\boldsymbol{n}_{23},\\
\boldsymbol{u}_{23} &= \tfrac{1}{8}a_{23}\boldsymbol{n}_{23}.
\end{aligned}$$

Dann führt die $l-$Gleichung der Galerkin-Approximation für die Kontinuitätsgleichung (3.51) zur Cell-Vertex-FVM auf dem Gebiet $S = \bigcup_{K\in NT(l)} K$ (Abb. 3.9), das den Träger der linearen Ansatzfunktion ψ_l bildet, genauer

$$\sum_{i\in I}\int_K \psi_l\,(\nabla\varphi_i\cdot\boldsymbol{u}_i)\;d\boldsymbol{x} = \sum_{A_i,A_j\in\partial S}\bar{\boldsymbol{u}}_{ijl}\cdot\boldsymbol{n}_{ij}\,a_{ij} \approx \sum_{K_l\in NT(l)}\int_{\partial K_l}\boldsymbol{u}\cdot\boldsymbol{n}\,do = \int_{\partial S}\boldsymbol{u}\cdot\boldsymbol{n}\,do \tag{3.55}$$

wobei $\bar{\boldsymbol{u}}_{ijl} = \frac{1}{8}(\boldsymbol{u}_i+\boldsymbol{u}_j+3\boldsymbol{u}_{ij}+3\boldsymbol{u}_{il}+3\boldsymbol{u}_{jl})$. Das kann aber als keine FV-Formulierung der Kontinuitätsgleichung betrachtet werden, da sich solche Träger paarweise jeweils in zwei Dreiecken überdecken.

P_2/P_0-Stokes-Element

Das P_2/P_0-Stokes-Element hat eine Konvergenzrate erster Ordnung, wird aber nicht so häufig benutzt wie die zuvor betrachteten Elemente, da die Geschwindigkeits-Ansatzfunktionen zweiter Ordnung sind. Die exakte Integration der GFEM kann aber ähnlich zur P_1^M/P_0-Element-FVM formuliert werden.

Im Unterschied zum Element P_1^M/P_0 werden die Ansatzfunktionen φ_i zweiter Ordnung behandelt, die mit der Bezeichnung der Abbildung 3.7 im Anhang (Gleichungen A.12) vollständig aufgeschrieben sind. Die exakte Integration der Galerkin-Formulierung des Diffusionsterms ist allerdings, wie erwartet, linear und mit Hilfe der Formeln (3.28),(3.32) und (3.33) auf dem Dreieck $A_1A_2A_3$ liefert

$$\begin{aligned}
\sum_{i\in I}\int_K \nabla\varphi_i\nabla\varphi_1 d\boldsymbol{x}\,\boldsymbol{u}_i &= \tfrac{a}{2b}\boldsymbol{u}_1 + \tfrac{c}{6}\boldsymbol{u}_2 + \tfrac{a-bc}{6b}\boldsymbol{u}_3 - \tfrac{2c}{3}\boldsymbol{u}_{12} - \tfrac{2a-2bc}{3b}\boldsymbol{u}_{13}\\
&= \left(\tfrac{1}{2}\boldsymbol{u}_1 + \tfrac{1}{6}\boldsymbol{u}_2 - \tfrac{2}{3}\boldsymbol{u}_{12}\right)\cot\alpha + \left(\tfrac{1}{2}\boldsymbol{u}_1 + \tfrac{1}{6}\boldsymbol{u}_3 - \tfrac{2}{3}\boldsymbol{u}_{13}\right)\cot\beta\\
&= \tfrac{2(3\boldsymbol{u}_1-4\boldsymbol{u}_{12}+\boldsymbol{u}_2)}{3a_{1,12}}m^K_{1,12} + \tfrac{2(3\boldsymbol{u}_1-4\boldsymbol{u}_{13}+\boldsymbol{u}_3)}{3a_{1,13}}m^K_{1,13},
\end{aligned} \tag{3.56}$$

$$\begin{aligned}
&\sum_{i\in I}\int_K \nabla\varphi_i\nabla\varphi_{12} d\boldsymbol{x}\,\boldsymbol{u}_i =\\
&= -\tfrac{2c}{3}\boldsymbol{u}_1 - \tfrac{2c}{3}\boldsymbol{u}_2 + \tfrac{4}{3}\left(\tfrac{a}{b} + \tfrac{b(1+c^2)}{a} - c\right)\boldsymbol{u}_{12} + \tfrac{4}{3}\left(c - \tfrac{b(1+c^2)}{a}\right)\boldsymbol{u}_{13} - \tfrac{4(a-bc)}{3b}\boldsymbol{u}_{23}\\
&= -\tfrac{2}{3}(\boldsymbol{u}_1+\boldsymbol{u}_2)\cot\alpha - \tfrac{4}{3}\boldsymbol{u}_{23}\cot\beta - \tfrac{4}{3}\boldsymbol{u}_{13}\cot\gamma + \tfrac{4}{3}(\cot\alpha+\cot\beta+\cot\gamma)\boldsymbol{u}_{12}\\
&= \tfrac{4(\boldsymbol{u}_{12}-\boldsymbol{u}_1)}{3a_{1,12}}m^K_{1,12} + \tfrac{4(\boldsymbol{u}_{12}-\boldsymbol{u}_2)}{3a_{2,12}}m^K_{2,12} + \tfrac{4(\boldsymbol{u}_{12}-\boldsymbol{u}_{13})}{3a_{13,12}}m_{13,12} + \tfrac{4(\boldsymbol{u}_{12}-\boldsymbol{u}_{23})}{3a_{23,12}}m_{23,12},
\end{aligned} \tag{3.57}$$

was den zum Dreieck K gehörenden Anteil der Differenzen-Approximation des Randintegrals darstellt. Die Druckterme stimmen mit denselben Termen (3.44),(3.45) des Elementes P_1^M/P_0 bis auf eine positive Konstante überein

$$\begin{cases} \displaystyle\int_K \frac{\partial\varphi_1}{\partial x}\,dxdy\,p^K = 0\,p^K & = \frac{2}{3}(m_{1,12}^K\tilde{n}_{1,12}^x + m_{1,13}^K\tilde{n}_{1,13}^x)\,p^K \\ \displaystyle\int_K \frac{\partial\varphi_1}{\partial y}\,dxdy\,p^K = \frac{a}{6}\,p^K & = \frac{2}{3}(m_{1,12}^K\tilde{n}_{1,12}^y + m_{1,13}^K\tilde{n}_{1,13}^y)\,p^K, \end{cases} \tag{3.58}$$

$$\begin{cases} \displaystyle\int_K \frac{\partial\varphi_{12}}{\partial x}\,dxdy\,p^K = -\frac{2b}{3}p^K = & \frac{4}{3}(m_{12,23}\tilde{n}_{12,23}^x + m_{12,13}\tilde{n}_{12,13}^x \\ & +m_{12,1}^K\tilde{n}_{12,1}^x + m_{12,2}^K\tilde{n}_{12,2}^x)\,p^K, \\ \displaystyle\int_K \frac{\partial\varphi_{12}}{\partial y}\,dxdy\,p^K = \frac{2(a-bc)}{3}p^K = & \frac{4}{3}(m_{12,23}\tilde{n}_{12,23}^y + m_{12,13}\tilde{n}_{12,13}^y \\ & +m_{12,1}^K\tilde{n}_{12,1}^y + m_{12,2}^K\tilde{n}_{12,2}^y)\,p^K. \end{cases} \tag{3.59}$$

Diese Konstante kommt auch im Diffusionsterm (3.56)-(3.57) entsprechend vor. Die Summe von (3.56),(3.57) und (3.58),(3.59) über die Dreiecken $K \in NK(k)$ liefert nun die Beziehungen (3.61) und (3.62), die zusammen eine FVM für die Impulsgleichungen darstellen.

Da die Ansatzfunktion für den Druck konstant ist, enthält die exakte Integration der GFEM der Kontinuitätsgleichung dieselben Glieder, wie die Druckterme. Allerdings werden sie in diesem Fall durch die Kanten des Dreiecks K ausgedrückt, um disjunkte Kontrollvolumen zu erhalten:

$$\begin{aligned} \sum_{i\in I}\int_K \left(\frac{\partial\varphi_i}{\partial x}u_i + \frac{\partial\varphi_i}{\partial y}v_i\right)\,dxdy &= 0u_1 - \tfrac{b}{6}u_2 + \tfrac{b}{6}u_3 - \tfrac{2b}{3}u_{12} + \tfrac{2b}{3}u_{13} + 0u_{23} \\ &\quad +\tfrac{a}{6}v_1 - \tfrac{bc}{6}v_2 - \tfrac{a-bc}{6}v_3 + \tfrac{2(a-bc)}{3}v_{12} + \tfrac{2bc}{3}v_{13} - \tfrac{2a}{3}v_{23} \\ &= -\bar{u}_{12}a_{12}\sin\beta + \bar{u}_{13}a_{13}\sin\alpha \\ &\quad +\bar{v}_{12}a_{12}\cos\beta + \bar{v}_{13}a_{13}\cos\alpha - \bar{v}_{23}a, \end{aligned} \tag{3.60}$$

wobei diesmal $\bar{u}_{ij} = \frac{u_i+4u_{ij}+u_j}{6}$ und $\bar{v}_{ij} = \frac{v_i+4v_{ij}+v_j}{6}$ $(i,j=1,2,3,\ i\neq j)$ gilt. Sinus- und Kosinus-Funktionen sind nach (3.43) die Komponenten des äußeren Normalvektors zu den Dreieckkanten. Die Summe über alle Freiheitsgrade bildet vollständige Randintegral-Approximationen, wobei die Impulsgleichungen auf dem Voronoi-Diagramm $\mathcal{V}_{h/2}$ (Abb. 3.8,a)

$$\begin{cases} \displaystyle\int_{\partial V_k} \nabla\boldsymbol{u}\cdot\tilde{\boldsymbol{n}}\,do = \frac{2}{3}\sum_{i\in NK(k)} \frac{\boldsymbol{u}_i - 4\boldsymbol{u}_{ik} + 3\boldsymbol{u}_k}{a_{ik}}m_{k,ik} + O(h^2) & (A_k \notin \mathcal{T}_{h/2}) \\ \displaystyle\int_{\partial V_{jk}} \nabla\boldsymbol{u}\cdot\tilde{\boldsymbol{n}}\,do = \frac{4}{3}\sum_{i\in NK'(jk)} \frac{\boldsymbol{u}_{jk} - \boldsymbol{u}_i}{a_{i,jk}}m_{i,jk} + O(h^2) & (A_k \in \mathcal{T}_{h/2}), \end{cases} \tag{3.61}$$

$$\int_{\partial V_k} p\tilde{\boldsymbol{n}}\,do = C\sum_{j\in NT(k)} p_j\tilde{\boldsymbol{n}}\,|\partial V_k^{K_j}| + O(h), \qquad C = \begin{cases} \frac{2}{3}, & A_k \notin \mathcal{T}_{h/2} \\ \frac{4}{3}, & A_k \in \mathcal{T}_{h/2} \end{cases} \tag{3.62}$$

und die Kontinuitätsgleichung auf der Triangulierung $\mathcal{T}_h$ (Abb. 3.8,b)

$$\int_{\partial K_l} \boldsymbol{u}\cdot\boldsymbol{n}\,do = \sum_{A_i,A_j\in K_l} \frac{\boldsymbol{u}_i + 4\boldsymbol{u}_{ij} + \boldsymbol{u}_j}{6}\cdot\boldsymbol{n}_{ij}\,a_{ij} + O(h) \tag{3.63}$$

formuliert werden, $k = 1, ..., N_u$, $l = 1, ..., N_p$. Die Approximation entspricht einer Node-Centered-FVM für die Impulsgleichungen und einer Cell-Centered-FVM für die Kontinuitätsgleichung.

P_2/P_1-Taylor-Hood-Element

Das bekannte und häufig benutzte Stokes-Element wurde in 1973 von Taylor und Hood [158] eingeführt und später von Brezzi und Falk [25] untersucht. Das verallgemeinerte Taylor-Hood-Element P_m/P_{m-1} ($m \geq 2$) erfüllt die LBB-Bedingung (3.26) und konvergiert mit der Rate $O(h^m)$ auch im dreidimensionalen Fall [21, 22].

Wir interessieren uns insbesondere für die niedrige Ordnung des Elements $m = 2$. Für die GFEM des Diffusionsterms (3.36) gilt die Approximation aus 3.61, da die Ansatzfunktionen für die Geschwindigkeit $\varphi_i \in P_2$ dieselben wie im Element P_2/P_0 sind. Die Geschwindigkeits-Druck-Kopplung ist sehr ähnlich zum P_1^M/P_1-Element. Druck-Terme haben dieselbe Gestalt wie in (3.50) mit den Druck-Approximationen

$$\bar{p}_{ijk} = \frac{2p_i}{3}, \qquad \bar{p}^*_{ijk} = \frac{2p_i + p_j + p_k}{3}. \tag{3.64}$$

Die Kontinuitätsgleichung kann genau so in der Form wie in (3.55) dargestellt werden, wobei

$$\bar{\boldsymbol{u}}_{ijl} = \frac{\boldsymbol{u}_{ij} + \boldsymbol{u}_{il} + \boldsymbol{u}_{jl}}{6}. \tag{3.65}$$

Die vollständigen Integrationsformeln und deren Umwandlungen befinden sich im Anhang A.1.

Aus den gleichen Gründen, wie das P_1^M/P_1-Element, kann das P_2/P_1-Element in keine FVM umgewandelt werden, deshalb besitzt dieses Element keine lokale Konservativitätseigenschaft.

Stabilisierung mit der Blasen-Funktion

Diese Stabilisierungs-Methode kommt in mehreren bekannten Stokes-Elementen vor. Wir wählen das sog. Mini-Element P_1^B/P_1 und interessieren uns zuerst für die Approximation des Diffusionsterms. Jede Geschwindigkeits-Ansatzfunktion $\varphi \in P_1$ bekommt zusätzlich einen Summanden, die sog. Blasen-Funktion, aus dem Raum

$$B_3 = \{\varphi_B \in H_0^1(\Omega) | \varphi_{B|K} = C\, b_K,\ C \in \mathbb{R},\ b_K = \lambda_1\lambda_2\lambda_3\}, \tag{3.66}$$

wobei λ_i ($i = 1, 2, 3$) lineare Polynome sind, die auf Seiten des Dreiecks K verschwinden. Außer drei Geschwindigkeits- und Druck-Freiheitsgraden in den Ecken, hat das Mini-Element noch einen vierten Geschwindigkeits-Freiheitsgrad im geometrischen Schwerpunkt des Dreiecks, der in der Basis $\{a, b, c\}$ (Abb. 3.6) die Koordinaten $(\frac{2a-bc}{3}, \frac{b}{3})$ hat. Dann erhält man die Blasen-Funktion in der Form

$$\varphi_B = C\left(\frac{1}{ab}xy - \frac{a-bc}{ab^2}y^2 - \frac{1}{a^2b}x^2y + \frac{a-2bc}{a^2b^2}xy^2 + \frac{c(a-bc)}{a^2b^2}y^3\right). \tag{3.67}$$

Die Integration des Diffusionsterms ergibt

$$\sum_{i=1}^{4} \int_K \nabla\varphi_i \nabla\varphi_1 \, d\boldsymbol{x} \, \boldsymbol{u}_i = \begin{array}{l} \frac{1}{2}\cot\beta(\boldsymbol{u}_1 - \boldsymbol{u}_2) + \frac{1}{2}\cot\alpha(\boldsymbol{u}_1 - \boldsymbol{u}_3) \\ + \frac{C}{180}(\cot\alpha + \cot\beta + \cot\gamma)(\boldsymbol{u}_1 + \boldsymbol{u}_2 + \boldsymbol{u}_3 + \boldsymbol{u}_4). \end{array} \tag{3.68}$$

Im Vergleich zum Satz 3.1 und (3.34) schließt die letzte Approximation noch einen zusätzlichen symmetrischen Summanden ein (zweite Zeile in 3.68), der offensichtlich unsymmetrischerweise auf zwei dualen Voronoi-Kanten m_{12}^K und m_{13}^K nicht zerlegt werden kann, um eine FVM herzuleiten.

Zusammenfassung

Wir haben die oft benutzte LBB-stabile Stokes-Elemente niedriger Ordnung auf Delaunay-Triangulierungen in zwei Dimensionen betrachtet und versucht eine äquivalente FV-Formulierung zu finden. Für die Stokes-Elemente mit konstantem Druck ist das gelungen. Diese besitzen genau wie auch andere Elemente mit unstetigem Druck also eine lokale Konservativitätseigenschaft. Wir haben aber nicht nur diesen bekannten Fakt wieder geprüft, sondern die äquivalenten FV-Formulierungen explizit bekommen (Tab. 3.1). Diese FVM erben die Stabilität- und Konvergenz-Eigenschaften der entsprechenden FE-Approximation.

Im 3D-Fall ist eine Durchführung von denselben Umformulierungen auf der Grundlage einer geometrischen Basis in einem Tetraeder wesentlich komplizierter. Man kann aber vermuten, dass dieselbe äquivalente FV-Approximationen auch im 3D-Fall aufgeschrieben werden können, da alle verwendeten geometrischen Eigenschaften die Analogbeziehungen im 3D-Fall haben.

Für ein zweidimensionales Viereckgitter ist ein ähnliches Beispiel der Äquivalenz im Buch von Gresho und Sani [70], Abschnitt 3.14 für das Q_1/Q_0 dargestellt, obwohl das Element theoretisch instabil ist.

Term	P_1^M/P_0	P_2/P_0	
Diffusion	$\sum_{i\in NK'(k)} \frac{\boldsymbol{u}_i - \boldsymbol{u}_k}{a_{ik}} m_{ik}$	$\frac{2}{3}\sum_{i\in NK(k)} \frac{\boldsymbol{u}_i - 4\boldsymbol{u}_{ik} + 3\boldsymbol{u}_k}{a_{ik}} m_{k,ik}$	$A_k \notin \mathcal{T}_{h/2}$
		$\frac{4}{3}\sum_{i\in NK'(jk)} \frac{\boldsymbol{u}_{jk} - \boldsymbol{u}_i}{a_{i,jk}} m_{i,jk}$	$A_{jk} \in \mathcal{T}_{h/2}$
Druck	$\sum_{j\in NT'(k)} p_j \tilde{\boldsymbol{n}} \, \lvert\partial V_k^{K_j}\rvert$	$\frac{2}{3}\sum_{j\in NT(k)} p_j \tilde{\boldsymbol{n}} \, \lvert\partial V_k^{K_j}\rvert$	$A_k \notin \mathcal{T}_{h/2}$
		$\frac{4}{3}\sum_{j\in NT(k)} p_j \tilde{\boldsymbol{n}} \, \lvert\partial V_k^{K_j}\rvert$	$A_k \in \mathcal{T}_{h/2}$
Kontinuität	$\sum_{A_i, A_j \in K} \frac{\boldsymbol{u}_i + 2\boldsymbol{u}_{ij} + \boldsymbol{u}_j}{4} \boldsymbol{n}_{ij} a_{ij}$	$\sum_{A_i, A_j \in K} \frac{\boldsymbol{u}_i + 4\boldsymbol{u}_{ij} + \boldsymbol{u}_j}{6} \boldsymbol{n}_{ij} \, a_{ij}$	

Tabelle 3.1: FEM/FVM-äquivalente Approximationen

Kapitel 4

Generative Software-Komponenten

The purpose of computing is insight,
not numbers.
- R.W. Hamming

Generative Programmierung (GP) beschreibt Design und Implementierung von Software-Komponenten, die zur Generierung von spezialisierten und hochoptimierten Systemen mit bestimmten Anforderungen kombiniert werden können [43]. Dabei wird angestrebt [37, 38, 39]:

- die konzeptionelle Lücke zwischen Programm-Code und Domänen-Konzepten zu verringern,
- eine hohe Wiederverwendbarkeit und Anpassungsfähigkeit zu erreichen,
- die Verwaltung von vielen Komponenten-Versionen zu vereinfachen,
- die Effizienz sowohl nach CPU-Zeit als auch nach Speicherbedarf zu erhöhen.

Die GP verlagert den Fokus von einem einzelnen Softwaresystem auf eine Softwaresystemfamilie, um die richtigen Komponenten für mehrere mögliche Softwaresysteme aus dem Problembereich (auch Domäne) berücksichtigen zu können. Eine Anforderungsspezifikation wird dabei als Eingabe benutzt, um ein gut angepasstes und optimiertes Zwischen- oder Endprodukt, d.h. ein Mitglied einer Softwaresystemfamilie, automatisch aus elementaren wiederverwendbaren Komponenten mit Hilfe von Konfigurationswissen zu generieren [38]. Dafür wurden für die GP folgende Grundsätze entwickelt:

- Trennung von Anteilen (engl. separation of concerns): Man betrachtet nur ein Problem in einem Zeitpunkt. Jedes Problem wird in einzelne Codeteile getrennt, die dann zur Generierung notwendiger Komponente kombiniert werden;
- Parametrisierung von Unterschieden: Das erlaubt Komponenten-Mengen kompakt zu repräsentieren;
- Analyse und Modellierung von Abhängigkeiten und Wechselwirkungen: Die jeweiligen abhängigen oder unabhängigen Parameter werden festgestellt. Solche Parameter-Abhängigkeiten heißen auch das horizontale Konfigurationswissen, wenn sie auf einem Abstaktionsniveau vorkommen;

- Trennung des Problemraums (spezifische Begriffe, Merkmale, Verfahren) vom Lösungsraum (zu implementierende generische Komponenten). Die Abbildung dazwischen wird mit Hilfe verschiedener Programmiertechniken, wie z.B. Template-Metaprogrammierung (Abschnitt 4.1), mit dem vertikalen (Parameter aus unterschiedlichen Abstraktionsniveaus) Konfigurationswissen hergestellt;
- Domänenspezifische Optimierung: Wenn die Komponenten statisch (in der Kompilierungszeit) konfiguriert sind, kann zusätzlicher Aufwand von unbenutztem Code, Laufzeit-Überprüfung, usw. eliminiert werden. Genau so kann eine komplizierte spezielle Optimierung, wie Loop-Unrolling (Abschnitt 5.3.3), durchgeführt werden.

Die GP integriert Aspekte von vielen Zugängen wie Metaprogrammierung (Abschnitt 4.1), generische Programmierung, objektorientierte Programmierung, aspektorientierte Programmierung und Domain-Engineering (Kap. 5).

Für die numerische Strömungssimulation als eine aufwendige Rechenaufgabe hat die GP einen großen Vorteil: Man kann einen Strömungssimulator entwickeln, der sowohl über viele numerische Verfahren verfügt als auch eine hohe Rechen-Performance zeigen kann. Mittels der statischen Konfiguration können richtige Kombinationen von numerischen Verfahren für unterschiedliche Simulatorteile gewählt werden.

Die Ideen der GP lassen sich in unterschiedliche Programmiersprachen, wie z.B. Lisp, CLOS, Smalltalk, realisieren, aus welchen nur die Sprache C++ gleichzeitig für wissenschaftliche Berechnungen aktiv eingesetzt wird.

In diesem Kapitel wollen wir uns ausschließlich auf die in der Strömungssimulation auftretenden statischen Parameter konzentrieren. Diese Parameter beinhalten diskrete Navier-Stokes-Gleichungen (Abschnitt 4.2), polynomiale Ansatzfunktionen auf einem Referenzelement (Abschnitt 4.3), gemischte Stokes-Elemente (Abschnitt 4.4). Sie werden in kleinere Parameter zerlegt, wie z.B. Aufgabendimension, Geometrie-Typ, usw. und mittels Template-Metaprogrammierung implementiert. Die Einteilung in die statischen (vor der Kompilierung bekannten) und dynamischen (erst in der Laufzeit bekannten) Daten sowie Merkmaldiagramme werden im nächsten Kapitel vorgestellt.

4.1 Metaprogrammierung in C++

Metaprogrammierung bedeutet Entwicklung von Programmen, die mit anderen Programmen oder mit sich selbst als mit eigenen Daten manipulieren oder führen Teilaufgaben während der Kompilierungszeit durch, die anderenfalls in der Laufzeit ausgeführt werden [186]. In der Programmiersprache C++ ist die Metaprogrammierung durch sog. Templates möglich und wird deshalb Template-Metaprogrammierung genannt. Sie kann in allen Aspekten der Definition mit den zur Kompilierungszeit bekannten Daten angewendet werden: statische Programm-Konfiguration [38], Modellierung und Berechnung von mathematischen Formeln und Objekten [122, 123]. In diesem Abschnitt werden zuerst Grundlagen sowie Anweisungen und Datenstrukturen zusammengefasst, die die Template-Metaprogrammierung zu einer vollständigen Metasprache machen.

4.1.1 Templates

Die Templates wurden 1990 im Buch von Elle und Stroustrup [44] eingeführt. Sie stellen parametrisierte Typen bereit und bilden dadurch die Grundlage für die generische und die Metaprogrammierung. Es gibt zwei Arten von Templates: Klassen-Templates und Funktionen-Templates. Ein Klassen-Template beschreibt eine Menge von Klassen durch Variation von Template-Parametern, z.B. das Klassen-Template `ClassTemplate` mit den Parametern `T` und `N`:

```
template<class T, int N>
class ClassTemplate {
   // eine Klassen-Definition mittels Typ T und Konstante N
};
```

Die Templates haben zwei folgende wichtige Eigenschaften:

1. ***Instanziierung:*** Erzeugung einer Klasse oder einer Funktion aus dem Template mit bestimmten Template-Parametern. Dieses geschieht immer, wenn ein Template in einem Programm benutzt werden soll, z.B.

   ```
   ClassTemplate<char,5> a;
   ```

2. ***Spezialisierung:*** Verdeutlichung einer Template-Definition durch Definition mit vollständig oder partiell angegebenen Template-Parametern. Partielle Spezialisierung ist nur für Klassen-Templates erlaubt, deshalb sind sie flexibler als Funktionen-Templates, z.B.

   ```
   template<int N>
   class ClassTemplate<char,N> {
      // Eine andere Definition mittels Konstante N
   };
   ```

Sowohl bei der Instanziierung als auch bei der Spezialisierung können beliebige eingebaute und neudefinierte Datentypen bzw. Klassen und Klassen-Templates auftreten. Ein besonderes Verfahren ist ein Klassen-Template zu sich selbst als Template-Parameter zu geben und dadurch eine Art von Rekursion zu erzeugen. Dieses Verfahren wird noch häufig beim Aufbau von Datenstrukturen benutzt (Abschnitt 4.1.3,4.3). Eine weitere Template-Dokumentation findet man im Buch von Bjarne Stroustrup [156], Kap. 13.

Bei einem Standardisierungstreffen im Jahr 1994 hat Erwin Unruh ein Programm gezeigt, das gar nicht kompiliert wurde, sondern den Compiler gezwungen hat, die Primzahlen in den Fehlermeldungen auszugeben [165]. Etwas später im Jahr 1995 hat Todd Veldhuizen [170] das nun klassische Beispiel der Fakultätsberechnung in der Kompilierungszeit veröffentlicht und mit der Idee von Erwin Unruh gezeigt, dass die Template-Metaprogrammierung Turing-vollständig ist. Dieser Artikel war die Basis, auf der die Template-Metaprogrammierung aufgebaut wurde. Wir betrachten ein ähnliches Beispiel, das die ganzzahlige Potenz X^N in der Kompilierungszeit berechnet (Listing 4.1). Die Anwendung des Ausdrucks `IPow<X,N>::value` in einem Programm instanziiert auch `IPow<X,N-1>::value` usw. bis die Schlußspezialisierung von `IPow<X,0>` erreicht und der Ausdruck ausgewertet ist. Die Zahlen X und N sind statische Konstanten, d.h. sie müssen vor der Kompilierung explizit angegeben

sein. Anstelle von `IPow<X,N>::value` im übersetzten Programmcode bleibt nur das berechnete Ergebnis als Zahl und es entsteht kein zusätzlicher Rechenaufwand.

```
template<int X, unsigned int N>
struct IPow {
   enum { value=X*IPow<X,N-1>::value };
};

template<int X>
struct IPow<X,0> {
   enum { value=1 };
};
```

Listing 4.1: Berechnung der ganzzahligen Potenz in der Kompilierungszeit

Ohne Absicht und unerwartet hat die Einführung von Templates durch die rekursive Instanziierung und Spezialisierung von Templates zu einer Metasprache im Rahmen der Sprache C++ geführt.

4.1.2 Anweisungen

In diesem und nächsten Abschnitt wollen wir die Aussage unterlegen, dass ein beliebiger Algorithmus als Metaprogramm implementiert werden kann. Dafür benötigt man drei Sprachkonstruktionen: sequenzielle Ausführung, Auswahl und Schleife.

Ein Metaprogramm wird immer als ein oder mehrere Klassen-Templates implementiert und nur dann ausgeführt, wenn eines von diesen Klassen-Templates instanziiert wird. Deshalb geschieht die Ausführung von Metaprogrammen genau so, wie die entsprechende Instanziierung im Programmtext vorkommt.

Die Auswahl wird in Metaprogrammen mit Hilfe der Spezialisierung implementiert (Tab. 4.1). Das entsprechende Klassen-Template wird für den Fall spezialisiert, wenn der betreffende Ausdruck erfüllt ist. Sonst gilt die allgemeine Klassen-Template-Definition. Die ganzen Zahlen werden durch `enum`-Anweisung behandelt (Tab. 4.1, 1. Spalte), wobei durch statische Funktionen (Tab. 4.1, 2. Spalte) auch Gleitkommazahlen berechnet werden können.

Metaprogrammierung		C-Sprache

`template<int A>`
`class IfElse {`
`  enum { B=2 };`
`};`
`template<>`
`class IfElse<0> {`
`  enum { B=1 };`
`};`
	`template<int A>`
	`class IfElse {`
	`  static B() { return 2;}`
	`};`
	`template<>`
	`class IfElse<0> {`
	`  static B() { return 1;}`
	`};`
		`if (A==0)`
		`  B=1;`
		`else`
		`  B=2;`

Tabelle 4.1: Beispiel einer `if-else`-Anweisung der Metaprogrammierung

Metaprogrammierung	C-Sprache

`template<int N, int I=1>`
`class ForLoop {`
`  enum { S=2*I+ForLoop<N,I+1>::S };`
`};`
`template<int N>`
`class ForLoop<N,N> {`
`  enum { S=0 };`
`};`
	`int i,s=0;`
	`for (i=1; i<n; ++i)`
	`{`
	`  s += 2*i;`
	`}`

Tabelle 4.2: Beispiel einer `for`-Schleife der Metaprogrammierung

Eine Metaschleife wurde bereits im Beispiel der Potenzberechnung (Listing 4.1) eingesetzt. Die allgemeine Idee einer Metaschleife liegt in der Rekursion über einen Template-Parameter, der auch als Zähler dieser Schleife angesehen werden kann. Die Rekursion bzw. die Schleife muss durch eine Spezialisierung bezüglich dieses Zählers beendet sein. Tabelle 4.2 stellt eine for-Schleife dar, die die doppelten Werte des Zählers i, $i = \overline{1, n-1}$ aufsummiert. Der Template-Parameter I bekommt 1 als Defaultwert und muss bei der Instanziierung nicht gegeben werden, wenn kein anderer Wert gewünscht ist.

4.1.3 Datenstrukturen

Obwohl die im letzten Kapitel vorgestellte Metaprogrammierungs-Techniken schon seit dem Artikel von Veldhuizen [170] bekannt sind, wurden sie nur auf ganz einfache Anwendungen begrenzt, wie z.B. die ganzzahlige Potenzberechnung in Listing 4.1. Die Ursache besteht darin, dass neben der erhöhten Syntax-Komplexität von einfachen Anweisungen auch keine geeignete Datenstrukturen vorhanden waren, die häufig bei allgemeinen Aufgaben benötigt werden. Ein Metaprogramm muss auch z.B. ein Array, eine Matrix oder noch kompliziertere Daten behandeln. Eine Lösung stellen rekursive Klassen-Templates bereit. Ein Beispiel dafür sind Typlisten, die von Alexandrescu [3, 4] entwickelt wurden. Das Klassen-Template `Typelist` hat zwei Parameter als Datentypen (Tabelle 4.3). An erster Stelle wird einen Typ eingetragen, der in der Liste als statische Daten gespeichert werden soll. Die zweite Stelle besetzt entweder ein nächstes `Typelist`-Template, das die Liste fortsetzt, oder die leere Klasse `NullType`, die das Ende der Liste kennzeichnet. So entsteht ein Array von Typen einer theoretisch beliebigen Länge. Ganz ähnlich kann ein ganzzahliges Array definiert werden [38, 123]. Das Klassen-Template `Numlist` bekommt eine ganze Zahl als ersten Template-Parameter und hat denselben `NullType` als Endmarke. Eine Array-Definition kann wie folgt aussehen:

```
typedef Numlist<5,Numlist<-3,Numlist<7,NullType>>> NewArray;
```

Die Daten eines Metaprogramms sind also ganze Zahlen und Datentypen. Vielleicht werden auch Gleitkommakonstanten in Zukunft als Templates-Parameter erlaubt. Die ganzen Zahlen werden durch die `enum`-Anweisung gespeichert und behandelt, wobei berechnete Gleitkommawerte über statische Funktionen zurückgegeben werden. Datentypen werden mit dem `typedef`-Schlüsselwort definiert und können

nicht geändert werden. Stattdessen werden neue Datentypen durch die bereits existierenden Datentypen während des Ablaufs eines Metaprogramms erstellt. Konkrete Beispiele dafür sind im Abschnitt 4.2 angegeben.

Typelist	Numlist

`template<class U, class T>`
`struct Typelist {`
`  typedef U Head;`
`  typedef T Tail;`
`};`
	`template<int N, class T>`
	`struct Numlist {`
	`  enum { Num = N };`
	`  typedef T Tail;`
	`};`

Tabelle 4.3: Typelist- und Numlist-Klassen-Templates

Damit die Nutzung von solchen rekursiven Datenstrukturen einfacher ist, werden die grundlegenden Zugriffs- und Änderungs-Operationen als Meta-Algorithmen implementiert. Tabelle 4.4 listet die wesentlichen dieser Operationen aus [3] auf. Außer ähnlichen grundlegenden Operationen besitzen Numlisten auch rein mathematische Algorithmen (Tab. 4.5).

Aus den Typlisten und Numlisten können kompliziertere Datenkonstruktionen gebildet werden, z.B. eine Typliste von Numlisten, da die Numlisten selbst Datentypen sind; eine Typliste von Typlisten, usw. (Abschnitt 4.2-4.4).

Aufruf	**Beschreibung**
`Append<TList,T>::Result`	fügt einen Typ oder eine Typliste `T` in die Typliste `TList` hinzu
`Erase<TList,T>::Result`	löscht das erste Vorkommen von Typ `T` in `TList`
`Replace<TList,T,U>::Result`	ersetzt das erste Vorkommen von `T` mit `U` in `TList`
`TypeAt<TList,Index>::Result`	liefert den auf dem Platz `Index` stehende Typ aus `TList`
`IndexOf<TList,T>::value`	liefert die Nummer des ersten Vorkommens von `T` in `TList`
`Length<TList>::value`	liefert die Länge der Typliste `TList`

Tabelle 4.4: Meta-Algorithmen mit Typlisten

4.1.4 Von Meta- zu Laufzeit-Daten

Wir müssen nun eine allgemeine Brücke von der Metaprogrammierung zur Laufzeit-Programmierung bilden, da nicht nur einzelne statische Konstanten, sondern auch komplizierte Daten (Typlisten und Numlisten) in ein Laufzeit-Programm übergeben werden müssen. Dafür dienen bestimmte Klassen-Templates, die zwar mit derselben Rekursion über Template-Parameter erzeugt werden, außerdem eine oder mehrere Funktionen (statische oder nicht) enthalten, die auch Funktionsargumente besitzen. Solche Funktionen können gleichzeitig sowohl zu statischen Daten eines Metaprogramms als auch zu Laufzeitdaten über Funktionsargumente verfügen.

Aufruf	Beschreibung
`Max<NList>::value`	liefert das maximale Element der Liste `NList`
`Min<NList>::value`	liefert das minimale Element der Liste `NList`
`AddConst<NList,N>::Result`	addiert `N` zu jedem Element von `NList`
`AddAt<NList,I,N>::Result`	addiert `N` zum Element `I` von `NList`
`Add<NList1,NList2>::Result`	addiert zwei Listen
`MultConst<NList,N>::Result`	multipliziert `N` mit jedem Element von `NList`
`MultAt<NList,I,N>::Result`	multipliziert `N` mit dem Element `I` von `NList`
`Mult<NList1,NList2>::Result`	multipliziert zwei Listen
`Sum<NList>::value`	berechnet die Summe aller Elementen
`Sort<NList>::Result`	sortiert die Liste `NList` aufsteigend

Tabelle 4.5: Meta-Algorithmen mit ganzzahligen Listen (Numlist)

Wir betrachten ein Beispielprogramm, das Daten aus einem `Numlist` einem Array zuweist (Listing 4.2). Das Klassen-Template `ReturnNumList` hat keine allgemeine Definition und ist nur für zwei Fälle spezialisiert: für eine Numliste oder für die Endmarke `NullType`. Im ersten Fall wird die statische Funktion `apply` rekursiv aufgerufen. Im zweiten Fall ist die Funktion `apply` leer, da das Ende der Liste erreicht ist.

```
1  template<class NList> struct ReturnNumList;

3  template<int N, class Tail>
   struct ReturnNumList<Numlist<N, Tail> > {
5     static void apply(int* array) {
         *array = N;
7        ReturnNumList<Tail>::apply(array+1);
      }
9  };

11 template<>
   struct ReturnNumList<NullType> {
13    static void apply(int*) { }
   };
```

Listing 4.2: Übergabe von Metadaten in ein Array

Ein anderes und schon klassisches Beispiel ist die sog. Expression-Templates-Technik, die eine effiziente Lösung (ohne temporäre Objekte) von Linear-Algebra-Ausdrücken mit Hilfe der Operator-Überladung darstellt [96, 169, 171, 172]. Die überladenen mathematischen Operatoren $*, /, +, -$ bilden aus einem Linear-Algebra-Ausdruck einen binären Baum von Klassen-Templates, wobei linker und rechter Zweig den linken bzw. rechten Operator-Argument speichert. Die entsprechend erzeugten Objekte speichern Referenzen (nicht die ganzen Objekte) auf die Laufzeit-Daten (Vektor, Matrix oder Vektor-Matrix-Ausdruck). Nach der Auflösung des Ausdrucks erreicht der Compiler die Stelle, wo das Ausdrucks-Resultat zugewiesen werden muss. An der Stelle wird das Resultat (Vektor oder Matrix) über den gespei-

cherten Baum ohne temporäre Vektoren/Matrizen ausgewertet.

Zusammenfassung

Entsprechend den vorhergehenden Abschnitten erhalten wir die Template-Metaprogrammierung als eine vollständige Metaprogrammiersprache sowohl für numerische Berechnungen kleiner Komplexität als auch für die Behandlung von Teilprogrammen in der Form von Klassen-Templates mit den folgenden Vor- und Nachteilen:

+ Kompilierungszeit-Berechnungen mit den statischen Daten reduzieren die Rechenzeit bei jedem nachfolgenden Aufruf des Programms.
− Kompilierungszeit-Berechnungen benötigen viel mehr Zeit und Speicher als die Analog-Laufzeitberechnung, da viele Klassen-Templates dabei instanziiert werden müssen. Deshalb sind solche Berechnungen mit nur relativ kleiner Dimension und Komplexität möglich.
+ Statische Konfiguration des Codes: Nur die durch Templates-Parameter ausgewählten Klassen werden kompiliert. Der Code enthält dann nur die notwendigen Algorithmen ohne dynamische Bindung, überflüssige Laufzeit-Auswahl, usw.
+ Der generierte Code ist effizient bzgl. CPU-Zeit durch die statische Bindung, da alle Templates nach der Kompilierung aufgelöst sind.
− Komplizierte Syntax macht die Entwicklung von Metaprogrammen schwerer als diejenige von normalen Programmen.

Außerdem gibt es keine Tools zur Fehlersuche in Metaprogrammen. Man kann allerdings den Compiler zwingen, notwendige Mitteilungen als Kompilierungsfehler auszudrucken. Das Listing 4.3 stellt ein Metaprogramm dar, das eine Typliste durchläuft und am Ende der Liste, in der Spezialisierung für `NullType` einen Fehler erzeugt, da dort kein Typ `Result` definiert wird. Der Compiler druckt die ganze Liste in den Fehlermeldungen aus. Das Programm erzeugt nur dann keinen Fehler, falls die Typliste leer ist, d.h. nur den `NullType` enthält. Wie man andere verständliche Compiler-Meldungen generieren kann, findet man bei Alexandrescu [3].

```
template<class TList> struct Print;

template<> struct Print<NullType> { };

template<class Head, class Tail>
struct Print<Typelist<Head, Tail> > {
   typedef typename Print<Tail>::Result Result;
};
```

Listing 4.3: Ausdruck einer rekursiven Typliste in der Kompilierungszeit

4.2 Modellierung der Navier-Stokes-Gleichungen

4.2.1 Voraussetzungen

Ein Softwaresystem benutzt normalerweise eine einzige oder begrenzte Zahl von diskreten Gleichungen, die in der Laufzeit nicht geändert werden. Deshalb kann man diese Gleichungen in der Kompilierungszeit modellieren. Wir betrachten die instationären Navier-Stokes-Gleichungen in der Form (3.1) ohne Randbedingungen, die jedoch in der Laufzeit definiert und behandelt werden müssen. Eine diskrete Form der Impuls-Gleichung wird in drei Schritten gewonnen (Abschnitt 3.1.1):

1. Zeitdiskretisierung mittels der Finiten-Differenzen-Methode;
2. Linearisierung des nichtlinearen konvektiven Terms in der Impulsgleichung mit Hilfe der Newtonmethode;
3. Vereinfachung der diskreten Gleichung, da nach Schritt 1 und 2 ähnliche Konvektionsglieder auftreten können.

Für die ersten zwei Schritte existieren unterschiedliche Methoden (Abschnitt 3.1), die als Meta-Algorithmen implementiert werden können, falls die Gleichungen zuerst als Meta-Daten dargestellt werden. Kombinationen dieser Methoden liefern unterschiedliche diskrete Schemata, die auch mit einem Meta-Algorithmus vereinfacht werden können (3. Schritt).

4.2.2 Implementierung

Diskrete Zeit-Schemata bestehen aus den Gliedern der ursprünglichen Gleichungen auf einem aktuellen Zeitschritt n und vorhergehenden Zeitschritten. Zum Aufbau eines diskreten Schema sind also der Zeitschritt und die ganzzahlige Konstante eines Summanden der Navier-Stokes-Gleichungen wichtig. Deshalb kann man die Summanden als leere Klassen-Templates mit ganzzahligen Parametern für ein Metamodell definieren (Listing 4.4). Alle Klassen-Templates für lineare Terme besitzen

```
template<int Coef, int Step> class Time { };
template<int Coef, int Step> class Diffusion { };
template<int Coef, int Step1, int Step2> class Convection { };
template<int Coef, int Step> class Pressure { };
template<int Coef, int Step> class Force { };
template<int Coef, int Step> class Continuity { };
```

Listing 4.4: Definition von Termen der Navier-Stokes-Gleichungen durch Klassen-Templates

zwei Parameter: der ganzzahlige Koeffizient und der Zeitschritt, wobei für die nichtlineare Konvektion zwei Zeitschritte angegeben werden müssen. Diese Gleichungs-Glieder müssen nun in einem Gleichungs-Modell zusammengesetzt werden, das mit Hilfe einer Typliste (Abschnitt 4.1.3) entsprechend der Impuls-Gleichung implementiert wird (Listing 4.5). Das ist das Ausgangsmodell ohne Zeitdiskretisierung und Linearisierung. Der aktuelle Zeitschritt n wird durch 0 und jeder vorhergehende

Zeitschritt durch $-1, -2, ...$ in diesem Modell bezeichnet. Einige Linearisierungs-Methoden benötigen auch den Anfangszeitschritt, der als 1 angenommen werden kann. Die Viskosität und andere mögliche Fluid-Eigenschaften sind in diesem Mo-

```
typedef Typelist<Diffusion<1,0>,
        Typelist<Convection<1,0,0>,
        Typelist<Pressure<-1,0>,
        Typelist<Force<-1,0>,NullType> > > > MomentumEquation;
```

Listing 4.5: Ein diskretes Modell der Impulsgleichung durch Klassen-Templates

dell unwichtig, da sie erst in der Laufzeit berücksichtigt werden sollen und nicht verloren gehen, solange wir wissen, zu welchem Gleichungsterm sie gehören.

Zur Umsetzung von Zeitdiskretisierungs- und Linearisierungs-Algorithmen sind noch Hilfs-Algorithmen notwendig, die hier ohne vollständige Implementierung aufgelistet sind (Tab. 4.6). Eine diskrete Gleichung (wie in Listing 4.5) wird in diesen Algorithmen durch den Parameter `Equat` übergeben.

Aufruf	**Beschreibung**
`ConstMult<Equat,N>::Result`	multipliziert mit `N` alle Koeffizienten in `Equat`
`Shift<Equat,N>::Result`	addiert `N` zu allen Zeitschritten in `Equat`
`Simplify<Equat>::Result`	vereinfacht `Equat` mathematisch

Tabelle 4.6: Meta-Algorithmen zur Modellierung von diskreten Impulsgleichungen

Als Beispiel-Algorithmus zur Zeitdiskretisierung implementieren wir das Einschritt-θ-Schema (3.2). Den Wert von $\theta = \frac{a}{b}$ geben wir durch einen Bruch an, der alle gängigen abgeleiteten Schemata überdeckt. Der Algorithmus (Listing 4.6) bekommt drei Parameter: die ursprüngliche Gleichung `Equation`, Zähler `Numer` und Nenner `Denom` für die Bestimmung von θ. Die ursprüngliche Gleichung (wie in Listing 4.5) auf dem aktuellen Zeitschritt n wird zuerst mit a multipliziert (Hilfsalgorithmus `ConstMult`), dazu wird dieselbe mit $(b - a)$ multiplizierte Gleichung auf den Zeitschritt $n - 1$ addiert (Algorithmen `Shift` und `Append`). Als letzte wird die mit b multiplizierte Diskretisierung der Zeitableitung ins Schema hinzugefügt.

Auf eine ähnliche Weise wird die Newton-Linearisierung umgesetzt, wobei der nichtlineare Term $\boldsymbol{u}^n \cdot \nabla \boldsymbol{u}^n$ mit der Approximation über den vorhergehenden Zeitschritt $\boldsymbol{u}^n \cdot \nabla \boldsymbol{u}^{n-1} + \boldsymbol{u}^{n-1} \cdot \nabla \boldsymbol{u}^n - \boldsymbol{u}^{n-1} \cdot \nabla \boldsymbol{u}^{n-1}$ in der diskreten Gleichung ersetzt wird. Der Algorithmus (Listing 4.7) bekommt die Gleichung `Equation` und der zu diskretisierende nichtlineare Term `Conv` und benutzt den Algorithmus `Replace` (Tab. 4.4), um den Konvektionsterm mit der Approximationsformel zu ersetzen.

Die eingeführten Algorithmen müssen konsequent das Impulsgleichungs-Modell (wie in Listing 4.5), zuerst die Zeitdiskretisierung, danach die Linearisierung, abarbeiten. Dabei können ähnliche Konvektionsglieder entstehen, deshalb wird auf das Resultat noch der Algorithmus `Simplify` angewendet (Listing 4.8). Als Beispiel ist $\theta = \frac{1}{2}$ gewählt. Das Resultat ist ein diskretes Schema für die Impulsgleichung, das als eine Typliste aufgebaut ist.

Damit das Modell einer diskreten Gleichung einheitlich ist, wird die aus einem einzigen Divergenzterm bestehende Kontinuitätsgleichung auch als eine Typliste mit

```
template<class Equation, unsigned int Numer, unsigned int Denom>
class ThetaMethod {
   typedef Typelist<Time<Denom,0>,
           Typelist<Time<Denom,-1>,NullType> > temp;
public:
   typedef typename Append<temp,typename Append<
           typename ConstMult<Equation,Numer>::Result,
           typename ConstMult<
           typename Shift<Equation,-1>::Result,Denom-Numer>
                    ::Result>::Result>::Result Result;
};
```

Listing 4.6: Metaprogramm der Einschritt-Theta-Methode zur Zeitdiskretisierung der Impulsgleichung

```
template<class Equation, class Conv> struct NewtonMethod;

template<class Equation, int Coef, int Step,
         template<int, int, int> class Conv>
class NewtonMethod<Equation, Conv<Coef,Step,Step> > {
   typedef Conv<Coef,Step,Step> T0;
   typedef Typelist<Conv<Coef,Step,Step-1>,
           Typelist<Conv<Coef,Step-1,Step>,
           Typelist<Conv<-Coef,Step-1,Step-1>,NullType> > > T1;
public:
   typedef typename Replace<Equation,T0,T1>::Result Result;
};
```

Listing 4.7: Metaprogramm der Newton-Methode zur Linearisierung der Navier-Stokes-Gleichungen

einem Element definiert. Damit ist auch die Möglichkeit beibehalten, dass die Kontinuitätsgleichung noch zusätzliche Stabilisierungsterme einschließen kann.

Die Navier-Stokes-Gleichungen sind ein Gleichungssystem. Um nicht nur zwei, sondern mehrere Gleichungen in einem System modellieren zu können, bilden wir ein System als Typliste von Gleichungen, d.h. auch von Typlisten. Beide Typlisten, der Impulsgleichung und der Kontinuitätsgleichung, werden in einer neuen Typliste, d.h. im Modell der Navier-Stokes-Gleichungen, mit eingeschlossen (Listing 4.8). Die in diesem Format generierten diskreten Schemata können nun als Template-Parameter zur Erstellung der lokalen Steifigkeitsmatrizen (Abschnitt 5.2.3) und zum Assemblierungsprozess (Abschnitt 5.2.4) übergeben werden.

Ähnlich zu den zwei dargestellten Beispielen (Listing 4.6 und 4.7) können noch viele weitere Zeitdiskretisierungs- und Linearisierungs-Methoden als Meta-Algorithmen, die diskrete Gleichungen als Typlisten bearbeiten, implementiert werden.

```
typedef typename Simplify<
        typename NewtonMethod<
        typename ThetaMethod<MomentumEquation,1,2>::Result
        ,Convection<1,0,0> >::Result >::Result DiscreteMomentum;
typedef Typelist<Continuity<-1,0>,NullType>
DiscreteContinuity;
typedef Typelist<DiscreteMomentum,
        Typelist<DiscreteContinuity,NullType> > DiscNavierStokes;
```

Listing 4.8: Ein diskretes Modell der Navier-Stokes-Gleichungen durch Klassen-Templates

4.2.3 Ausblick auf andere partielle Differentialgleichungen

Man kann leicht bemerken, dass mit den definierten Termen (Listing 4.4) auch alle Sonderfälle der Navier-Stokes-Gleichungen (wie in Listing 4.8) modelliert werden können, wie z.B. Stokes-, Euler-, Konvektions-Diffusions- und andere Gleichungen, wobei die Zeitdiskretisierung und/oder die Linearisierung je nach Gleichungstyp ausfallen soll. Das erhöht noch mehr die Vielfalt der Modelle diskreter Gleichungen durch die entwickelten wiederverwendbaren Software-Komponenten.

Die ursprüngliche Idee kann natürlich für andere Familien der partiellen Differentialgleichungen weiter entwickelt werden. Dafür muss man ein Abstraktionsniveau tiefer im entwickelten Gleichungsmodell gehen und den Aufbau der einzelnen Gleichungsterme genauer betrachten. Die Grundkomponenten jedes Terms sind bekannte oder unbekannte Vektor- oder Skalar-Größen und die Operatoren (Produkt, Ableitung, Integration), die im Prinzip auch durch Klassen-Templates definiert werden können. Auf dieser Basis können dann nicht nur Terme der Navier-Stokes-Gleichungen, sondern auch beliebige weitere Gleichungsterme gebildet werden.

4.3 Aufbau von Ansatzfunktionen

Wir betrachten hier Ansatzfunktionen (AF), die auf den Referenz-Elementen in 2D (Dreieck, Viereck) und in 3D (Tetraeder, Hexaeder) aufgebaut werden und dadurch von keinem bestimmten Gitter abhängig sind. Wenn die Dimension, die Ordnung, der AF-Typ und der Geometrie-Typ (Dreieck-, Viereck-Geometrie) schon in der Kompilierungszeit vorgegeben sind, kann eine AF durch Metaprogramme gebildet und behandelt werden. Das Endziel ist, die lokalen Element-Matrizen (siehe auch Abschnitt 5.2.3) für die bestimmten Glieder der inkompressiblen Navier-Stokes-Gleichungen zu bekommen.

4.3.1 Konforme Elemente

Die konformen GFEM-Approximationen werden durch in jedem Geometrie-Element stetige Polynomfunktionen aufgebaut und zeichnen sich dadurch aus, dass sie sich in jeder Dimension für jede Ordnung und jeden Geometrie-Typ algorithmisch behandeln lassen.

Abbildung 4.1: Ansatzfunktionen 1. und 2. Ordnung auf einem Dreieck

Algorithmischer Aufbau

Die allgemeine Form einer polynomialen AF zweiter Ordnung auf einem Dreieck ist

$$\hat{\varphi}_i(x,y) = a_0 + a_1 x + a_2 y + a_3 x^2 + a_4 xy + a_5 y^2. \tag{4.1}$$

In diesem Fall gibt es genau sechs Freiheitsgrade auf dem Dreieck zur Bestimmung der unbekannten Koeffizienten $a_0, ..., a_5$ (Abb. 4.2). Die Funktion $\hat{\varphi}_i$ ist gleich 1 im Punkt i und gleich 0 in den anderen Punkten des Dreiecks (Abb. 4.1). Daraus ergibt sich ein lineares Gleichungssystem bezüglich der Unbekannten $a_0, ..., a_5$, z.B. für die AF $\hat{\varphi}_0$, $(\hat{\varphi}_0(0,0) = 1)$:

$$\begin{aligned}
&\hat{\varphi}_0(0,0) = a_0 = 1\\
&\hat{\varphi}_0(\tfrac{1}{2},0) = a_0 + \tfrac{1}{2}a_1 + \tfrac{1}{4}a_3 = 0\\
&\hat{\varphi}_0(1,0) = a_0 + a_1 + a_3 = 0\\
&\hat{\varphi}_0(0,\tfrac{1}{2}) = a_0 + \tfrac{1}{2}a_2 + \tfrac{1}{4}a_5 = 0\\
&\hat{\varphi}_0(\tfrac{1}{2},\tfrac{1}{2}) = a_0 + \tfrac{1}{2}a_1 + \tfrac{1}{2}a_2 + \tfrac{1}{4}a_3 + \tfrac{1}{4}a_4 + \tfrac{1}{4}a_5 = 0\\
&\hat{\varphi}_0(0,1) = a_0 + a_2 + a_5 = 0
\end{aligned} \tag{4.2}$$

Statt ein solches Gleichungssystem für jeden Freiheitsgrad zu lösen, existiert eine günstigere Methode, die AF $\hat{\varphi}_i$ als ein Produkt linearer Polynome [52]

$$\hat{\varphi}_i(x,y) = C_i L_1 L_2 ... L_m \tag{4.3}$$

zu bilden, wobei $L_k = 0$, $k = \overline{1,m}$, die Gleichungen der Geraden in 2D oder Flächen in 3D sind, die alle Punkte (Freiheitsgrade) außer i durchlaufen. Wir definieren zuerst eine für beliebige Ordnung und Dimension geeignete Nummerierung von Freiheitsgraden, damit die Nummer i mit einem Freiheitsgrad auf einem Referenz-Element (Dreieck, Viereck in 2D; Tetraeder, Hexaeder in 3D) eindeutig zusammenhängt. Das kann ein Tupel in 2D (ein Tripel in 3D) sein, dessen Komponenten die Punkte entlang jeder Achse nummerieren (Abb. 4.2). Die AF $\hat{\varphi}_{1,0}$ $(\hat{\varphi}_{1,0}(\frac{1}{2},0) = 1)$ ist z.B.

$$\hat{\varphi}_{1,0}(x,y) = 4x(1-x-y), \tag{4.4}$$

wobei die Funktionen $x = 0$ und $1 - x - y = 0$ die AF $\hat{\varphi}_{1,0}$ entsprechend in den Punkten {0,0}, {0,1}, {0,2} und {0,2}, {1,1}, {2,0} schneiden und so annullieren. Der ganzzahlige Koeffizient garantiert, dass $\hat{\varphi}_{1,0}(\frac{1}{2},0) = 1$ erfüllt ist.

Wir können nun den AF-Aufbauprozess verallgemeinern und automatisieren. Im 3D-Fall gibt es folgende vier Arten linearer Polynome L_k für einen Tetraeder:

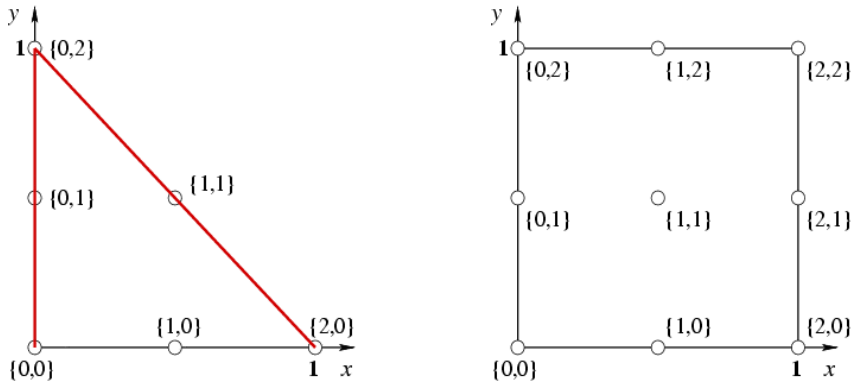


Abbildung 4.2: Einheits-Dreieck und -Quadrat mit durchnummerierten Freiheitsgraden zweiter Ordnung

a) Parallele zur Fläche yz: mx, $mx-1$, ..., $mx-m$;

b) Parallele zur Fläche xz: my, $my-1$, ..., $my-m$;

c) Parallele zur Fläche xy: mz, $mz-1$, ..., $mz-m$;

d) Diagonale: $1-mx-my-mz$, $2-mx-my-mz$, ..., $m-mx-my-mz$.

Beginnen wir mit der AF $\hat{\varphi}_{0,0,0}$. Sie besteht ausschließlich aus allen linearen Polynomen der Art (d). Die Bewegung zum nächsten auf der Achse x liegenden Punkt und dadurch der Aufbau der AF $\hat{\varphi}_{1,0,0}$ bedeutet den Ersatz des ersten linearen Polynoms der AF $\hat{\varphi}_{0,0,0}$ mit dem ersten Polynom der Art (a), d.h.

$$\hat{\varphi}_{1,0,0}(x,y,z) = \frac{1}{(m-1)!} mx(2-mx-my-mz)...(m-mx-my-mz). \quad (4.5)$$

Dann sieht die allgemeine Formel folgendermaßen aus:

$$\hat{\varphi}^{\triangle}_{i,j,k} = \frac{1}{i!}\underbrace{mx...(mx-i+1)}_{i}\ \frac{1}{j!}\underbrace{my...(my-j+1)}_{j}\ \frac{1}{k!}\underbrace{mz...(mz-k+1)}_{k}$$
$$\frac{1}{(m-i-j-k)!}\underbrace{(m-mx-my-mz)...(i+j+k+1-mx-my-mz)}_{m-i-j-k}. \quad (4.6)$$

Der Algorithmus für den Aufbau der AF $\hat{\varphi}_{i,j,k}$ auf einem Tetraeder besteht aus den folgenden Schritten:

1. In der Produkt-Darstellung (4.3) werden i,j,k Stück linearer Polynome der Art (a),(b),(c) entsprechend ausgewählt.
2. Die restlichen $L_{m-i-j-k}...L_m$ linearen Polynome werden der Art (d) in der umgekehrten Reihenfolge ausgewählt.
3. Damit die AF $\hat{\varphi}_{i,j,k}$ im Punkt $\{i,j,k\}$ gleich 1 ist, gilt es

$$C_i = \frac{1}{i!}\frac{1}{j!}\frac{1}{k!}\frac{1}{(m-i-j-k)!}.$$

Zum AF-Aufbau auf einem Hexaeder gibt es lineare Polynome nur aus den drei Arten (a),(b),(c). Die allgemeine Formel enthält m^3 lineare Polynome:

$$\begin{aligned}\hat{\varphi}^{\square}_{i,j,k} = \quad & \tfrac{1}{i!}\underbrace{mx...(mx-i+1)}_{i}\ \tfrac{1}{(m-i)!}\underbrace{(m-mx)...(i+1-mx)}_{m-i} \\ & \tfrac{1}{j!}\underbrace{my...(my-j+1)}_{j}\ \tfrac{1}{(m-j)!}\underbrace{(m-my)...(j+1-my)}_{m-j} \\ & \tfrac{1}{k!}\underbrace{mz...(mz-k+1)}_{k}\ \tfrac{1}{(m-k)!}\underbrace{(m-mz)...(k+1-mz)}_{m-k}.\end{aligned} \tag{4.7}$$

Der Algorithmus für den Aufbau der AF $\hat{\varphi}_{i,j,k}$ auf einem Hexaeder besteht aus den folgenden Schritten:

1. In der Produkt-Darstellung (4.3) werden i, j, k Stück linearer Polynome der Art (a),(b),(c) entsprechend ausgewählt.
2. Aus der Polynomlisten (a),(b),(c) werden die restlichen $m-i, m-j$ und $m-k$ linearer Polynome in der umgekehrten Reihenfolge mit -1 multipliziert ausgewählt. Dieses sichert auch das richtige Vorzeichen der AF.
3. Damit die AF $\hat{\varphi}_{i,j,k}$ im Punkt $\{i, j, k\}$ gleich 1 ist, muss gelten

$$C_i = \frac{1}{i!}\frac{1}{j!}\frac{1}{k!}\frac{1}{(m-i)!}\frac{1}{(m-j)!}\frac{1}{(m-k)!}.$$

Im 2D-Fall werden offensichtlich alle z-abhängigen Polynome in den Formeln (4.6,4.7) ausgelassen. Eine Verallgemeinerung auf den d-dimensionalen Fall ist ebenfalls ohne Probleme möglich.

Solche Ansatzfunktionen oder ihre partiellen Ableitungen werden in den linearen Gleichungsgliedern der GFEM paarweise multipliziert. Das Ergebnis muss dann über einem Referenz-Element integriert werden (Abschnitt 5.2.3). Sowohl Differentiation als auch Integration sind für allgemeinen Polynome nur in Standardform als Summe von Monomen $cx_1^{p_1}...x_d^{p_d}$ einfach durchzuführen. Die folgenden allgemeinen Integrations-Formeln eines Monoms über einem d-dimensionalen Tetraeder T_0 und Hexaeder H_0 können durch vollständige Induktion über d überprüft werden:

$$\int\cdots\int_{T_0} x_1^{p_1}...x_d^{p_d}dx_1...dx_d = \frac{p_1!...p_d!}{(p_1+...+p_d+d)!}, \tag{4.8}$$

$$\int\cdots\int_{H_0} x_1^{p_1}...x_d^{p_d}dx_1...dx_d = \frac{1}{(p_1+1)...(p_d+1)}. \tag{4.9}$$

Mit dem beschriebenen Verfahren werden alle konformen Lagrange-FEM-Approximationen P_m oder Q_m ausschließlich der Serendipity-Klasse (ohne die Punkte im Element-Inneren, siehe z.B. [152]) bedeckt. Das Verfahren beinhaltet die Behandlung von Monomen und Polynomen, deren Multiplikation, mathematische Vereinfachung, partielle Differenziation und Integration über einem Referenz-Element, die alle als Meta-Datenstrukturen und Meta-Algorithmen implementiert werden können. Im Fall der Lösung linearer Gleichungssysteme (wie z.B. 4.2) zur Bestimmung der AF-Koeffizienten a_i in (4.1), außer den genannten Operationen mit Polynomen, müssten die linearen Gleichungssysteme zusätzlich behandelt werden.

Implementierung

Für Modellierung eines Monoms $cx_1^{p_1}...x_d^{p_d}$ ist nur die Folge der ganzen Zahlen c, p_1, ... p_d wichtig. Diese Reihe kann durch eine Numliste (Abschnitt 4.1.3) dargestellt werden, wobei die erste Zahl immer die Konstante c bestimmt. Ein Polynom ist eine Summe von Monomen und kann ähnlich zur Modellierung von Gleichungen (Abschnitt 4.2) als eine Typliste, diesmal von Numlisten dargestellt werden. Wir benötigen noch eine weitere Meta-Datenstruktur für ein Produkt von Polynomen. Da ein Produkt mehrere Polynome enthalten kann, muss es auch durch eine Typliste von Polynom-Typlisten aufgebaut werden. Um eine Produkt-Typliste von einer Summe-Typliste zu unterscheiden wird für die Produkt-Typliste eine neue Endmarke `UnitType` eingeführt. So kann der Multiplikations-Algorithmus für diese unterschiedlichen Fälle spezialisiert werden. Eine konkrete Implementierung von Algorithmen ist aufwendiger im Vergleich zur Modellierung von Gleichungen (Abschnitt 4.2) und wird hier nur kurz klassifiziert:

- Aufbau einer AF als Produkt der linearen Polynome nach den Formeln (4.6,4.7), lineare Komplexität;
- Multiplikation von Polynomen und Monomen. Der Algorithmus führt die Produkt-Form in die Standardform über. Minimale Komplexität ist $O(2^m)$.
- Mathematische Vereinfachung eines Polynoms in der Standardform. Komplexität im schlechtesten Fall ist $O(2^{2m-1} - 2^{m-1})$. Multiplikation und Vereinfachung können zusammen schon bei dem Aufbau eingesetzt werden, um die Länge des entstehenden Polynoms zu verringern. Der Gewinn (Verlust) der Rechenzeit durch die verkürzte Länge (Einsatz der Multiplikation und der Vereinfachung) kann aber nur empirisch vorhergesehen werden.
- Partielle Differentiation eines Monoms bzw. Polynoms in der Standardform nach der Regel
$$\frac{\partial}{\partial x_i} cx_1^{p_1}...x_d^{p_d} = cp_i x_1^{p_1}...x_i^{p_i-1}...x_d^{p_d}.$$
- Integration über einem Referenz-Element nach den Formeln (4.8,4.9).

Grundlagen von diesen Meta-Algorithmen sind im Artikel [123] und im Anhang (A.4) vorgestellt.

Da die Komplexität der Multiplikations- und Vereinfachungs-Operationen nichtlinear ist, können solche Polynome nur kleiner Ordnung (die praktische Erfahrung zeigt $m < 4$) in der Kompilierungszeit erzeugt werden. Für die Ansatzfunktionen höherer Ordnung können die gleichen Algorithmen als Laufzeit-Routinen implementiert werden. Der größte Rechenaufwand liegt allerdings in der Berechnung der lokalen Steifigkeitsmatrizen (Abschnitt 5.2.3), wobei alle Ansatzfunktionen auf einem Referenz-Element paarweise multipliziert werden müssen.

4.4 Definition von Stokes-Elementen

Wir nehmen an, dass die Aufgaben-Dimension, der Gitter-Geometrie-Typ und die AF statische Daten sind. Dann können die gemischten Stokes-Elemente (Abschnitt 3.4) statisch konfiguriert werden. Ein gemischtes Stokes-Element besteht aus zwei

finiten Elementen entsprechend für die Geschwindigkeits- und die Druck-Diskretisierung. Jedes finite Element muss eine Art von Ansatzfunktionen beschreiben. Wenn wir uns auf die konformen Ansatzfunktionen beschränken, dann ist solche Beschreibung durch die Angabe der Dimension, des Geometrie-Typs und der Polynom-Ordnung vollständig (siehe auch die Merkmaldiagramme in der Abb. 5.7 und 5.8). Die Dimension $d = 1, 2, 3$ und die Ordnung $m = 0, 1, ...$ sind ganze Zahlen und können direkt als Template-Parameter definiert werden, wobei der Geometrie-Typ durch Klassen bestimmt wird (Listing 4.9).

Außer der angegebenen Information trägt die Element-Klasse auch ein Zeichen `c`, das zusammen mit dem Zeichen der Geometrie-Klasse (`Trian` oder `Quadr`) zur Erzeugung des Element-Namens dienen kann. In der Element-Klasse wird auch die statisch berechnete Anzahl der Freiheitsgrade M eines Referenz-Elementes bzw. die Dimension der lokalen Steifigkeitsmatrizen (Abschnitt 5.2.3) gespeichert. Auf einem Dreieckelement wird die Zahl M rekursiv bestimmt:

$$M(d, m) = \sum_{i=0}^{m} M(d-1, i), \qquad M(1, m) = m + 1, \tag{4.10}$$

wobei sie auf einem Viereckelement direkt berechnet wird:

$$M(d, m) = (m + 1)^d. \tag{4.11}$$

Da ein Stokes-Element aus genau zwei finiten Elementen besteht, können sie in einem gemeinsamen Klassen-Template zusammengefasst werden (Listing 4.10). Solche Klassen-Templates zusammen mit den Gleichungstermen (Abschnitt 4.2) sind dann Parameter für die Berechnung von lokalen Steifigkeitsmatrizen, die auch für kleine Ordnungen und lineare Gleichungsterme mit Hilfe der statischen Ansatzfunktionen (Abschnitt 4.3) innerhalb des Kompilierungsprozesses ausgewertet werden können.

```
struct Trian {
   enum { id=0, Char='P' };
};
struct Quadr {
   enum { id=1, Char='Q' };
};

template<class Geometry, unsigned char Dim, unsigned char Ord>
struct ConformElement {
   typedef Geometry Geom;
   enum { EType = CONFORM,
          Char  = 'c',
          Order = Ord,
          M     = DFnumber<Geom,Dim,Order>::value };
};
//z.B. Spezialisierung des P2-Elementes
template<unsigned char Dim>
struct P2 : public ConformElement<Trian,Dim,2> { };
```

Listing 4.9: Definition von konformen finiten Elementen durch Klassen-Templates

```
template<class VelocityElem, class PressureElem>
struct StokesElem : public VelocityElem,
                    public PressureElem {
  //Statische Neudefinitionen
};
//z.B. Spezialisierung des Taylor-Hood-P2/P1-Elementes
template<unsigned char Dim>
struct P2P1 : public StokesElem<P2<Dim>,P1<Dim> > { };
```

Listing 4.10: Definition von gemischten Stokes-Elementen durch Klassen-Templates

Kapitel 5

Software-Analyse, Design und Implementierung

Try to go with a static model when you can,
and rely on a dynamic model when you must.
- A. Alexandrescu

Die in Kapitel 3 beschriebenen FEM/FVM-Ansätze werden mit Hilfe der Meta- und objektorientierten Programmierung in ein flexibles und effizientes Software-Paket zur Simulation von inkompressiblen Strömungen umgesetzt. Die Software wird nach Richtlinien der Domain-Engineering-Theorie entwickelt und dargestellt [38, 190], damit die Komponenten des endgültigen Softwaresystems unabhängig sind und in Kombinationen unterschiedliche Aufgaben aus dem Problembereich lösen können. Das bedeutet, die Entwicklung des Softwaresystems wird als Entwicklung der gesamten Softwaresystemfamilie beschrieben, dann können viele Softwaresysteme aus den entwickelten und in hohem Maße wiederverwendbaren Komponenten zusammengestellt werden.

Der gesamte Simulations-Prozess besteht aus drei Schritten: Gitter-Generierung auf dem zu simulierenden Gebiet (Preprozessor), numerische Lösung der Navier-Stokes-Gleichungen auf dem erzeugten Gitter (Solver) sowie Text- und Grafik-Darstellung der Lösungs-Resultate (Postprozessor) (Abb. 5.1). Hinter jedem von diesen Schritten steht ein Forschungsgebiet mit eigenen Methoden und Fortschritten.

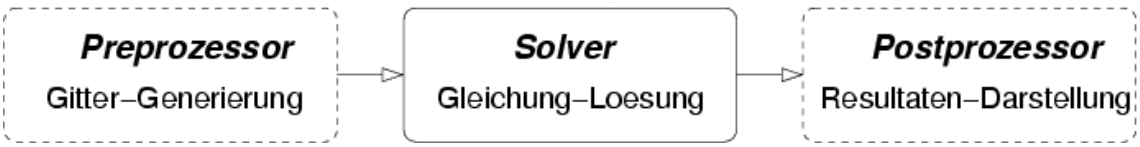


Abbildung 5.1: Strömungssimulationsmodell

Die hier betrachtete Softwarefamilie betrifft ausschließlich den Solver, obwohl sie vom Preprozessor und Postprozessor abhängt: Die Gitter-Daten müssen mindestens in einem Format vom Solver eingelesen werden; Die Resultate müssen in einem Daten-Format eines externen Postprozessors gespeichert werden. Wenn der Solver h-adaptiv ist, dann beinhaltet er auch Routinen der Gitter-Umgestaltung, deshalb

sind die Gitter-Datenstrukturen in diesem Fall sehr ähnlich zu den im Preprozessor verwendeten (Abschnitt 5.2.1).

Der Entwicklungsprozess entsprechend der Domain-Engineering-Theorie läuft in folgenden drei Etappen ab:

Analyse:	Definition von Anforderungen an ein Software-System im Problembereich (Abschnitt 5.2);
Design:	Aufbau einer gemeinsamen Architektur von Software-Systemen im Problembereich, Definition von Komponenten, die zu implementieren sind (Abschnitt 5.2);
Implementierung:	Umsetzung von wiederverwendbaren Komponenten, domänespezifischen Sprachen, Konfigurationsgeneratoren und Infrastruktur (Abschnitt 5.3).

Wir nennen den Problembereich bzw. die ***Domäne*** 'Strömungssimulations-Solver', den wir in diesem Abschnitt in Konzepte und Merkmale zerlegen und analysieren wollen. Die Beziehungen dazwischen werden in der Form von Merkmaldiagrammen (engl. feature diagrams [38]) dargestellt. Ein solches Domänenmodell hilft eine gemeinsame Architektur für die Softwaresysteme dieser Domäne zu entwickeln. Nach Hirschfeld [83] geht man von folgendem Grundverständnis der Architektur aus: 'Die Architektur eines Softwaresystems ist eine explizite Organisation. Sie bezeichnet die Dekompositionskomponenten und ihre Abhängigkeiten auf dem gröbsten Granularitätsniveau.'

Während des Entwicklungsprozesses werden naturgemäß Fehlentscheidungen getroffen, die erst zu einem späteren Zeitpunkt sichtbar werden. Zu diesem Zeitpunkt sind die Fehler jedoch noch korrigierbar, da sie noch nicht nach außen bekannt gemacht wurden. Um eine inkrementelle Verbesserung des Domänenmodells, der Architektur und der Implementierung zu erreichen, werden die drei Phasen mehrfach durchgelaufen. Hier wird jedoch nur die letzte Version des Domänenmodells und die darauf aufbauende Architektur dargestellt.

5.1 Statische vs. dynamische Daten

Die im Rahmen der Domain-Engineering-Theorie entwickelten Software-Komponenten werden statisch in der Kompilierungszeit mit Hilfe von Generatoren zusammengestellt, damit die wichtige Recheneffizienz nicht verloren geht. Im Vergleich zur GMCL-Matrix-Bibliothek ([125], [38],Kap.10), wobei eine Matrix nur einen Datenträger darstellt, können in einem Solver nicht alle Modell-Parameter statisch angenommen werden. Einige Daten werden erst in der Laufzeit geladen und können auch geändert werden, wie z.B. der Inhalt einer Matrix. Ein Solver beinhaltet viele Datenträger inklusive Matrizen unterschiedlicher Formate und transformiert dynamische Eingangsdaten in die Resultate.

Um die Vorteile der statischen Konfiguration und Kompilierungszeit-Berechnungen mittels Template-Metaprogrammierung (Abschnitt 4.1) vollständig auszunutzen, werden möglichst viele Parameter und Daten als statisch angenommen. Dann entsteht die folgende Unterteilung sämtlicher Eingangsdaten in zwei Klassen:

- ***Statische Daten*** sind vor der Kompilierung bekannt und können sowohl von Meta- als auch von Laufzeit-Algorithmen behandelt werden.

1. Dimension des Strömungsproblems $d = 1, 2, 3$.
2. Geometrie des Gitters: Dreieck- oder Viereck-Geometrie. Sie spielt für $d = 1$ keine Rolle. In 2D unterscheidet man Dreiecke oder Vierecke und in 3D - Tetraeder und Hexaeder. Andere 3D-Geometrie-Elemente können in Tetraeder und Hexaeder zerlegt werden.
3. Raumdiskretisierungsverfahren: FEM, FDM oder FVM. Im FEM-Kontext sind das aus Ansatzfunktions-Definitionen bestehende finite Elemente. Für die Navier-Stokes-Gleichungen werden sie paarweise zusammengesetzt.
4. Zu lösende diskrete Gleichung. Zeitdiskretisierungs- und Linearisierungs-Methoden für die Navier-Stokes-Gleichungen, die als Meta-Algorithmen implementiert werden können, liefern unterschiedliche diskrete Gleichungen (Abschnitt 4.2).

- ***Dynamische Daten*** stehen erst nach der Kompilierung zur Verfügung und werden nur von Laufzeit-Algorithmen behandelt.

  1. Die Raumdiskretisierung bzw. das Gitter.
  2. Randbedingungen, die sich auch in der Laufzeit ändern können.
  3. Anfangsbedingungen. Sie hängen von den Randbedingungen ab.
  4. Fluidparameter: Dichte, Viskosität und andere (siehe Abschnitt 2.1).
  5. Simulations-Resultate.

Es gibt natürlich mehrere Abhängigkeiten zwischen den statischen und dynamischen Daten, die auch zu den Verknüpfungen zwischen Meta- und Laufzeit-Programmen führen, das eine wichtige Implementierungsaufgabe mit den Techniken aus dem Abschnitt 4.1 ist. Konkrete Lösungen für einige Probleme dieser Art gibt es in den Abschnitten 5.2.3, 5.3.

5.2 Komponentenentwurf

Wir entwickeln ein Domänenmodell des Strömungssimulations-Solvers im Rahmen der Domain-Engineering-Theorie [38] und nutzen Merkmaldiagramme für die grafische Darstellung, deren Notation im Anhang (A.2.1) erklärt ist. Auf dem obersten Niveau enthält das Konzept **Solver** (Abb. 5.2) die folgenden Merkmale: Gitter, Anfangsbedingungen, zu lösende diskrete Gleichung, Matrix-Assemblierung und Resultate. Das Gitter wird von einem externen Datenformat importiert und wenn nötig auf mehrere Teile partitioniert (Abschnitt 5.2.1). Eine mit der FEM, FVM oder FDM (Abschnitt 5.2.3) diskretisierte Gleichung wird auf dem Gitter numerisch gelöst, wobei die Randbedingungen auf den importierten Gitter-Randgebieten definiert werden. Nach dem auf der Raumdiskretisierungsmethode basierten Assemblierungsprozess (Abschnitt 5.2.4) entsteht ein lineares Gleichungssystem, das mit den iterativen Verfahren (Abschnitt 5.2.5) und Präkonditionierungs-Techniken (Abschnitt 5.2.6) eine Approximationslösung des Strömungsproblems liefert. Jedes von diesen Merkmalen kann selbständig als ein Konzept mit eigenen Merkmalen betrachtet werden. Die wichtigsten von diesen wollen wir genauer betrachten.

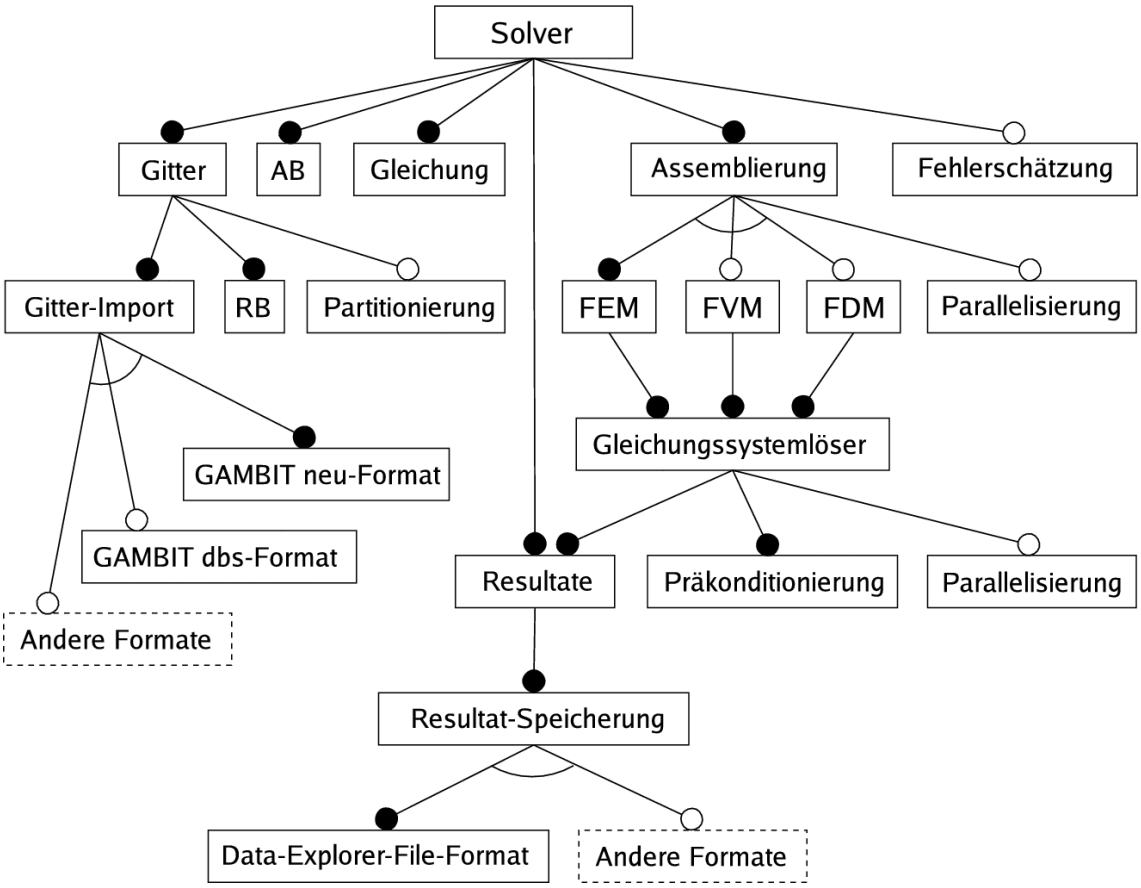


Abbildung 5.2: Merkmaldiagramm des Solver-Konzepts

5.2.1 Gitter

Das Konzept **Gitter** stellt eine Raumdiskretisierung dar, die durch Ecken und deren Zusammenhang als Kanten definiert ist. Dabei entstehende Gitter-Regionen oder auch Gitter-Zellen klassifizieren wir durch Dreiecke oder Vierecke im 2D-Fall und durch Tetraeder oder Hexaeder im 3D-Fall. Man betrachtet selten auch weitere mögliche 3D-Zellen: allgemeinere Pyramiden und Prismen. Das Konzept enthält aber keinen Parameter über den Geometrie-Typ (Abschnitt 5.3), da ein Gitter aus unterschiedlichen Zellen im allgemeinen Fall aufgebaut werden kann. Die Dimension ist jedoch konstant, wenn kein besonderes Strömungsproblem, wobei unterschiedliche Gitter-Teilgebiete diverse Dimensionen haben, berücksichtigt wird.

Gitter-Datenstrukturen

Um Bedürfnisse mehrerer Anwendungen mit einer Gitter-Datenstruktur abzudecken, repräsentiert man ein Gitter durch topologische Einheiten, die sog. Gitter-Entities. In drei Dimensionen sind das Ecken, Kanten, Flächen und Zellen; im 2D-Fall - Ecken, Kanten und Zellen; im 1D-Fall - nur Ecken und Zellen (Tab. 5.1). Alle 3D-Zellen haben Dreiecke und Vierecke bzw. 2D-Zellen als Seitenflächen. Dreiecke und Vierecke werden durch Kanten begrenzt, die selbst 1D-Zellen sind. Diese Gitter-Eigenschaft gestattet, eine Gitter-Datenstruktur von einer bis drei Dimensionen hierarchisch aufzubauen.

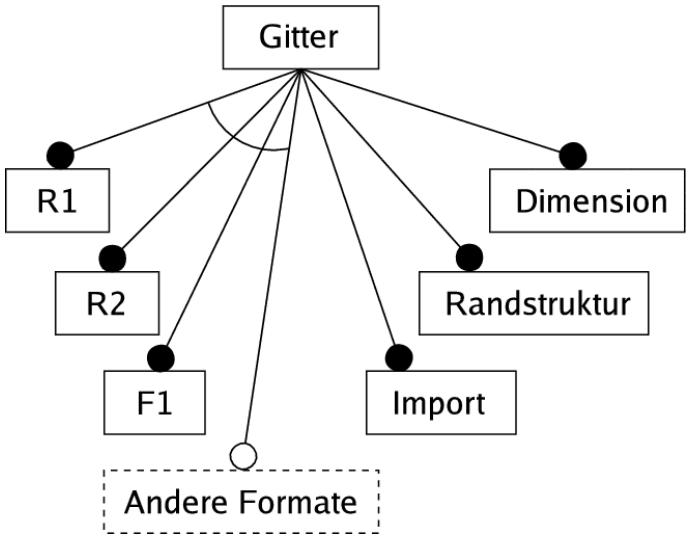
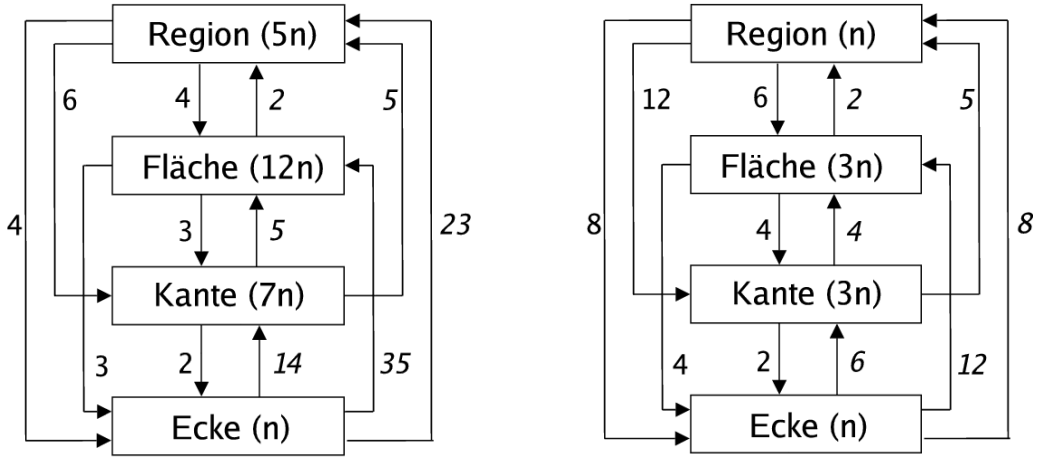


Abbildung 5.3: Merkmaldiagramm des Gitter-Konzepts

Bezeichnung	**Entity**	**engl. Bez.**	**Bedeutung**
R_i	Region	region	3D-Zelle
F_i	Fläche	face	2D-Zelle
E_i	Kante	edge	1D-Zelle
V_i	Ecke	vertex	

Tabelle 5.1: Hierarchie der Gitter-Komponenten

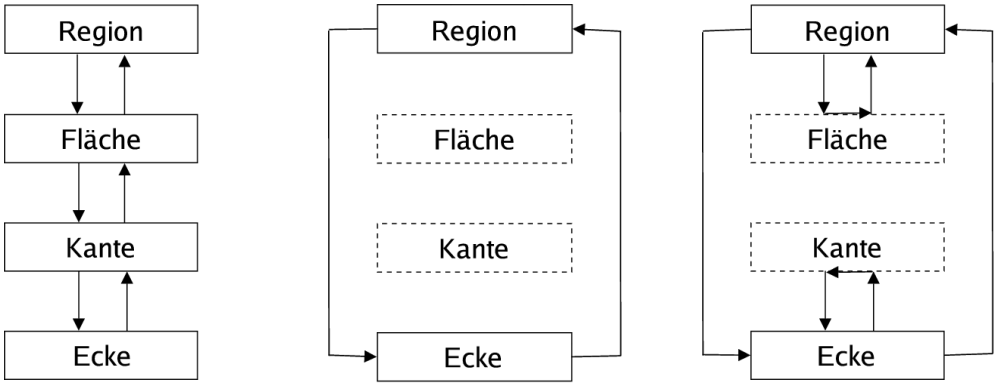


a) Tetraeder-Gitter b) Hexaeder-Gitter

Abbildung 5.4: Adjazenz-Beziehungen in einem 3D-Gitter

Eine Gitter-Entity (Ecke, Kante, Fläche, Zelle) besitzt zwei Adjazenz-Beziehungen, zu einem höherliegenden und einem darunterliegenden Entity-Niveau. In der Abbildung 5.4 sind alle mögliche Beziehungen dargestellt, die ein Gitter eindeutig beschreiben. In Klammern steht die durchschnittliche Anzahl von Entities bezüglich der Anzahl der Ecken n. Die Abwärtsbeziehungen haben feste Größe (gerade gedruckt), wobei für die Aufwärtsbeziehungen die durchschnittlichen Zahlen angegeben sind (kursiv gedruckt). Diese Zahlen ermöglichen Speicherbedarf und Algorith-

men-Komplexität unterschiedlicher Gitter-Datenstrukturen abzuschätzen. Die Datenstrukturen werden durch eine Untermenge von allen Entities und deren Beziehungen aus der Abbildung 5.4 gebaut (Abb. 5.5). Eine Gitter-Datenstruktur kann alle Entities und zwei Adjazenz-Beziehungen pro Entity enthalten. Obwohl eine solche vollständige Datenstruktur, wie z.B. in der Abbildung (5.5,F1), viel Speicher benötigt, können auf alle Entities in einem beliebigen Gitter-Algorithmus direkt zugegriffen werden. Bei einer Verfeinerung/Vergröberung eines Gitters mit dieser Datenstruktur müssen aber mehr Daten geändert werden als bei einer unvollständigen Datenstruktur (Abb. 5.5,R1,R2). Die in der Hierarchie dazwischenliegenden Entity-Niveaus können so ausgelassen werden, dass sie mit einem kleinen zusätzlichen Rechenaufwand aus den gespeicherten Entities bei Bedarf abgeleitet werden können.



F1: $(25M_e + 143M_c)n$ R1: $(6M_e + 43M_c)n$ R2: $(6M_e + 77M_c)n$

Abbildung 5.5: Beispiele für 3D-Gitter-Datenstrukturen und deren Speicher-Verbrauch

Es gibt keine universelle Gitter-Datenstruktur, die für alle Anwendungen gut geeignet ist. Eine vollständige Datenstruktur wird für ein besonderes Gitter mit gemischten Dimensionen oder ein nicht konformes Gitter, wie z.B. ein Gitter mit hängenden Knoten nach der adaptiven Verfeinerung, gebraucht. Für ein konformes Gitter ist eine unvollständige Datenstruktur sowohl nach Speicher-Verbrauch als auch nach durchschnittlicher Rechenleistung effizient. Innerhalb dieser zwei Klassen von Datenstrukturen werden mehrere Möglichkeiten betrachtet und diejenige ausgewählt, die für eine bestimmte Anwendung am besten geeignet sind. Die Datenstruktur R1 (Abb. 5.5) benötigt z.B. wenig Speicher und Operationen für die Erzeugung einer neuen Entity, wobei eine Nachbarnsuche aufwendiger als für Datenstrukturen F1 und R2 ist. Wir bezeichnen durch M_e und M_c den notwendigen Speicherplatz für eine Entity und eine Beziehung und nehmen dabei an, dass alle Entities und ihre Beziehungen den gleichen Speicherplatz brauchen. Dann ist der Speicherverbrauch der Datenstrukturen F1, R1, R2 mit den durchschnittlichen Adjazenz-Werten abgeschätzt in der Abbildung 5.4 unten angegeben. Ein detaillierter Vergleich von 10 unterschiedlichen und häufig gebrauchten Datenstrukturen wurde im Artikel von Garimella [63] durchgeführt.

Gitter-Algorithmen

Die Objekte Ecke, Kante, Fläche, Zelle werden durch die Symbole V_i, E_i, F_i, R_i bezeichnet (Tab. 5.1). Der logische Adjazenz-Operator ist durch Klammern () definiert. Dann liefert z.B. $V_i(R_j)$ die Ecken V_i einer Region R_j. Die Tabelle 5.2 stellt die Komplexität einer Nachbarnsuche-Operation für die drei Gitter-Datenstrukturen F1, R1, R2 aus der Abbildung 5.5 zusammen.

Typ	$F(R)$	$E(R)$	$V(R)$	$R(F)$	$E(F)$	$V(F)$	$R(E)$	$F(E)$	$V(E)$	$R(V)$	$F(V)$	$E(V)$
F1	4	36	30	2	3	13	50	5	2	619	399	14
R1	72	58	4	302	24	3	214	721	2	23	3462	1969
R2	84	58	4	2	24	3	214	731	2	23	3532	84

Tabelle 5.2: Komplexität der Nachbarnsuche von drei Tetraeder-Gitter-Datenstrukturen [63]

Obwohl bei einem Gitter-Datenstrukturen-Vergleich die Komplexität der Suche einer Entity bezüglich einer anderen Entity abgeschätzt wird, sind in der Praxis häufig Algorithmen notwendig, die die Entities des ganzen Gitters durchlaufen müssen. Die Komplexität eines solchen Algorithmus kann kleiner sein als die Summe von einzelnen Entity pro Entity Operationen, wenn insbesondere temporäre Hilfsdaten erzeugt werden. Für eine FEM-Implementierung wichtige Beispiele sind für solche Algorithmen folgende:

- Nummerierung von Freiheitsgraden eines Gitters. Eine Gitter-Zelle besitzt einen oder mehrere Punkte (Freiheitsgrade), wo die Approximationslösung einer PDG gesucht wird. Diese Punkte können sich in einer Gitter-Ecke, auf einer Kante, auf einer Fläche oder innerhalb der Zelle befinden und müssen eindeutig nummeriert werden. Ein möglicher effizienter Algorithmus ist im Abschnitt 5.3.2 erläutert.
- Bandbreitenreduzierende Umordnung von Freiheitsgraden. Die Idee des Cuthill-McKee-Algorithmus besteht darin, die entstehende Freiheitsgrad-Nummerierung so zu transformieren, dass die benachbarten Freiheitsgrade in der Reihenfolge möglichst nahe beieinander liegen. Dadurch platzieren sich die entsprechenden Elemente der Steifigkeitsmatrix nahe der Hauptdiagonale. Das bringt wesentliche Vorteile für direkte Verfahren zur Lösung des linearen Gleichungssystems. Obwohl man keinen offensichtlichen Vorteil der Umordnung für iterative Verfahren sieht, kann sie jedoch bei der Präkonditionierung, z.B. bei der unvollständigen ILU(0)-Zerlegung Vorteile bringen. Der Algorithmus wird häufig als eine Umordnung der Eckennummer erklärt [19, 132], kann aber genau so erfolgreich für die Freiheitsgrade angewendet werden, weil er auf der Graphen-Theorie basiert, wobei Freiheitsgrade/Ecken die Graphen-Ecken und ihre Nachbarschaft die Graphen-Kanten definiert. Zwei Freiheitsgrade i und j heißen benachbart, falls $\int_\Omega \varphi_i \varphi_j d\boldsymbol{x} \neq 0$ gilt. Der Algorithmus besteht aus folgenden Schritten [152]:

  1. Man berechne jeden Punktgrad. Der Grad eines Punktes (einer Ecke oder eines Freiheitsgrades) ist die Anzahl seiner Nachbarn.

2. Man finde die Nummer 1, d.h. den Punkt mit minimalem Grad. Geometrisch gesehen liegt dieser Punkt am Rand des Gebietes.
3. Zum ersten Punkt bestimme man alle benachbarten Punkten. Diese Punkte sollen mit zunehmenden Grad fortlaufend nummeriert werden. Zu jedem neu nummerierten Punkt bestimme man alle mit zunehmenden Grad benachbarten Punkte und nummeriere die, die noch nicht nummeriert sind. Man wiederhole diesen Schritt für alle weiteren Punkte.

Das wichtigste ist, dass jeder Gitter-Algorithmus eine lineare Komplexität besitzt, die von den anderen Gitter-Algorithmen nur durch eine von der Gitter-Größe unabhängige und kleinere Konstante unterscheiden darf. Im gegengesetzten Fall wird eine Behandlung von großen Gittern unmöglich.

Randbedingungen

Randbedingungen werden auf bestimmten Randgebieten des Gitters definiert. Diese Randgebiete bestehen aus Geometrie-Elementen einer Dimension kleiner als das Gitter selbst und müssen in der Gitter-Datenstruktur gespeichert werden. Für jede Aufgaben-Dimension kann der Rand durch eine Ecken-Menge gegeben sein. Dann müssen trotzdem die auf den Ecken basierten Randgebiete wieder hergestellt werden, um alle Rand-Freiheitsgrade berücksichtigen zu können. Die Randgebiete werden zellenbezogen behandelt, weil der Assemblierungsprozess (Abschnitt 5.2.4) über die Zellen ablaufen muss. Einerseits muss jede Rand-Entity (Ecke, Kante oder Fläche) seiner Zelle zugeordnet sein. Andererseits muss auch jede Zelle, die mindestens einen Randpunkt besitzt, in die Rand-Datenstruktur eingetragen werden. Deshalb kann diese Rand-Datenstruktur als eine Liste folgender Beziehungen dargestellt werden

$$c_i \longleftrightarrow (r_j, t_j), \qquad \begin{array}{l} i = 1, ..., N_u, \quad j = 1, ..., M_u, \\ t_j \in \{\text{Ecke, Kante, Fläche}\}. \end{array} \tag{5.1}$$

Die Zellen-Nummer c_i ist eindeutig im gesamten Gitter definiert, wobei die Rand-Entity-Nummer r_j nur die lokale Nummer des Geometrie-Elementes eines bestimmten Typs t_j ist. Auf einem unstrukturierten 3D-Gitter kann eine Zellen-Nummer mehrfach sowohl in einem als auch in mehreren Randgebieten auftreten, d.h. eine 3D-Zelle kann mehr als eine Rand-Fläche, die zu einem oder unterschiedlichen Randgebieten gehört, besitzen. Daraus folgt, dass es Ecken und Kanten geben kann, die sich auf unterschiedliche Randgebiete auch mit verschiedenen Randbedingungen gleichzeitig beziehen können. Solche Sonderfälle müssen besonders behandelt werden, indem diese Ecken/Kanten zu einem einzigen Randgebiet zugewiesen werden müssen, da sie sonst in die Steifigkeitsmatrix mehrfach assembliert werden.

Gitter-Ladung

Da die Gebiets-Vernetzung bzw. Gitter-Generierung außerhalb dieser Arbeit liegt, wird das Gitter von einem externen Gitter-Generator erzeugt und im zugehörigen Datenformat gespeichert. Dieses Datenformat muss vom Solver eingelesen und ins interne Format transformiert werden.

Partitionierung

Dieses Konzept ist die erste Etappe im Parallelisierungsprozess eines Strömungssimulations-Solvers. Das Gitter muss über mehrere Prozessoren mit den folgenden Bedingungen verteilt werden:

1. Die Anzahl von Gitter-Zellen muss auf jedem Prozessor gleich sein. Das Problem heißt Load-Balancing und ist insbesondere bei einer adaptiven Gitter-Verfeinerung aktuell, weil nach jedem Verfeinerungsschritt die Bilanz wieder effizienterweise erreicht werden muss;
2. Minimierung des totalen Kommunikations-Volumens;
3. Minimierung des maximalen Speicher-Volumens, das jeder Prozessor senden und empfangen muss;
4. Minimierung der Pakete-Anzahl, die jeder Prozessor senden und empfangen muss.

Um diese Bedingungen annähernd zu erfüllen, da die vollständige Lösung dieser komplizierten Optimierungsaufgabe zu aufwendig ist, wird das Gitter als ein Graph dargestellt. Im Gegensatz zum Cuthill-McKee-Algorithmus (Abschnitt 5.2.1) entsprechen die Gitter-Zellen den Graphen-Knoten und die Zellen-Nachbarschaft - den Graphen-Kanten. Zur Lösung der Graphen-Optimierungsaufgabe sind einige Programm-Pakete bzw. Programm-Bibliotheken vorhanden, wie z.B. Metis [88] und Jostle [182]. Beide nutzen eine ähnliche Vorgehensweise, das Multilevel-Verfahren. Da die Aufgabe auf dem ganzen Graphen zu aufwendig ist, wird sie zuerst auf einem gröberen Graphen gelöst und dann schrittweise für den verfeinerten Graphen präzisiert. Das Programm Metis nimmt die erste und die zweite als Hauptbedingungen. Die dritte und die vierte Bedingung sind insbesondere wichtig, wenn die Partitions-Anzahl hoch ist (über 30 Partitionen). Beim Metis kann auch die vierte Bedingung verlangt werden, dann ist die dritte Bedingung nur noch annähernd erfüllt.

5.2.2 Diskretisierte Navier-Stokes-Gleichungen

Die Implementierung des Konzepts **Gleichung** durch Klassen-Templates wurde im Abschnitt 4.2 ausführlich vorgestellt. Nun wollen wir die wichtigen Merkmale anhand des zugehörigen Merkmaldiagramms (Abb. 5.6) zusammenfassen.

Das Konzept besitzt die folgenden zwei Merkmale

- Die eindeutigen Gleichungsterme, für die die lokale Steifigkeitsmatrizen (Abschnitt 5.2.3) generiert werden müssen;
- Das aus den Termen entstehende diskrete Schema, das für die Navier-Stokes-Gleichungen aus der Impulsgleichung und der Kontinuitätsgleichung besteht. Auf die Impulsgleichung wird noch eine Zeitdiskretisierungs- und eine Linearisierungs-Methode angewendet.

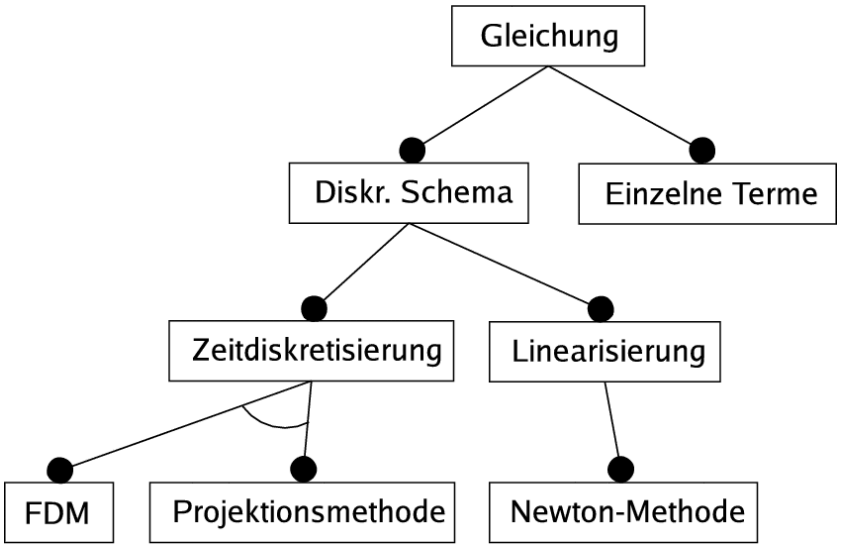


Abbildung 5.6: Merkmaldiagramm des Gleichungs-Konzepts

5.2.3 FEM und lokale Steifigkeitsmatrizen

Das Konzept **FEM** repräsentiert eine Raum-Approximation einer Differentialgleichung auf einem bestimmten Gitter. Wir betrachten genauer die FEM-Raum-Approximation, weil sie in vielen Fällen auch FDM und FVM als Sonderfälle beinhaltet (Abschnitt 3.5). Eine Standardstrategie zur Behandlung von unstrukturierten Gittern im FEM-Kontext ist, jedes Gitter-Element auf das Referenz-Element durch eine lineare Transformation $\Phi : K \to K_0$ abzubilden [2], z.B. in 2D:

$$\boldsymbol{x} = \Phi\boldsymbol{\xi} + \boldsymbol{f}_0, \qquad \Phi = \begin{pmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{pmatrix}, \quad \boldsymbol{f}_0 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \tag{5.2}$$

wobei $A_1(x_1, y_1)$, $A_2(x_2, y_2)$ und $A_3(x_3, y_3)$ die drei im Uhrzeigersinn geordneten Ecken eines Dreiecks oder Parallelogramms sind. Im 3D-Fall müssen die vier Ecken ein rechtsorientiertes kartesisches System bilden. Auf dem Referenz-Element K_0 wird die Näherungslösung in bestimmten Punkten gesucht, die auch Freiheitsgrade genannt werden. Jedem Freiheitsgrad gehört eine Ansatzfunktion $\hat{\varphi}_i$ und es gilt:

$$\varphi_i(\boldsymbol{x}) = \hat{\varphi}_i(\Phi^{-1}(\boldsymbol{x})), \qquad \frac{\partial \varphi_k}{\partial x_i} = \sum_{j=1}^{d} \frac{\partial \hat{\varphi}_k}{\partial \xi_j} \frac{\partial \xi_j}{\partial x_i}. \tag{5.3}$$

Im Fall der Navier-Stokes-Gleichungen müssen zwei unterschiedliche Sätze von Freiheitsgraden erstellt und behandelt werden, für die unbekannte Geschwindigkeit und den Druck. Dementsprechend gibt es zwei unterschiedliche Ansatzfunktionen $\hat{\varphi}_i$ und $\hat{\psi}_i$. Wir betrachten die Galerkin-Diskretisierung z.B. des semi-expliziten Euler-Schemas (3.3) in zwei Dimensionen integriert mit einem Geometrie-Element K mit

$\boldsymbol{x} = (x, y)^T$ und $\boldsymbol{u} = (u, v)^T$

$$\begin{aligned}
&\frac{1}{\triangle t}\sum_{i=1}^{M_u}\int_K \varphi_i\varphi_k d\boldsymbol{x}(u_i^n - u_i^{n-1}) + \nu\sum_{i=1}^{M_u}\int_K \nabla\varphi_i\nabla\varphi_k d\boldsymbol{x}\, u_i^n - \sum_{i=1}^{M_p}\int_K \psi_i\frac{\partial\varphi_k}{\partial x}d\boldsymbol{x}\, p_i^n \\
&+\sum_{i=1}^{M_u}\sum_{j=1}^{M_u}\int_K \varphi_i\frac{\partial\varphi_j}{\partial x}\varphi_k d\boldsymbol{x}\, u_i^{n-1}u_j^n + \sum_{i=1}^{M_u}\sum_{j=1}^{M_u}\int_K \varphi_i\frac{\partial\varphi_j}{\partial y}\varphi_k d\boldsymbol{x}\, v_i^{n-1}u_j^n = \int_K f_1^n\varphi_k d\boldsymbol{x}, \\
&\frac{1}{\triangle t}\sum_{i=1}^{M_u}\int_K \varphi_i\varphi_k d\boldsymbol{x}(v_i^n - v_i^{n-1}) + \nu\sum_{i=1}^{M_u}\int_K \nabla\varphi_i\nabla\varphi_k d\boldsymbol{x}\, v_i^n - \sum_{i=1}^{M_p}\int_K \psi_i\frac{\partial\varphi_k}{\partial y}d\boldsymbol{x}\, p_i^n \\
&+\sum_{i=1}^{M_u}\sum_{j=1}^{M_u}\int_K \varphi_i\frac{\partial\varphi_j}{\partial x}\varphi_k d\boldsymbol{x}\, u_i^{n-1}v_j^n + \sum_{i=1}^{M_u}\sum_{j=1}^{M_u}\int_K \varphi_i\frac{\partial\varphi_j}{\partial y}\varphi_k d\boldsymbol{x}\, v_i^{n-1}v_j^n = \int_K f_2^n\varphi_k d\boldsymbol{x},
\end{aligned} \tag{5.4}$$

wobei M_u bzw. M_p die Anzahl der Geschwindigkeits- bzw. Druck-Freiheitsgrade auf einem Element K bezeichnen. Diese Zahlen werden durch die Formel (4.10,4.11) bestimmt.

Wir interessieren uns an dieser Stelle ausschließlich für die in (5.4) auftretenden Integrale, weil jedes kompliziertere Zeitdiskretisierungs-/Linearisierungs-Schema nach der GFEM genau die gleichen Integrale als Koeffizienten enthält. Die Integrale unterscheiden sich nach einem bestimmten Term der ursprunglichen Gleichung: Zeitterm, Diffusionsterm, Konvektionsterm, Druckterm und Kontinuität. Jedes in (5.4) auftretende Integral wird auf das Referenz-Element mit Berücksichtigung von (5.3) abgebildet. Der erste Summand bzw. Zeitableitungsterm enthält die Geschwindigkeits-Massenmatrix

$$\int_K \varphi_i\varphi_k d\boldsymbol{x} = \det(\Phi)\int_{K_0}\hat{\varphi}_i\hat{\varphi}_k d\boldsymbol{\xi}, \qquad a_{ik} = \int_{K_0}\hat{\varphi}_i\hat{\varphi}_k d\boldsymbol{\xi}. \tag{5.5}$$

Das Integral in der rechten Seite kann gitterunabhängig ausgewertet und in eine lokale Element-Matrix $L = \{a_{i,k}\}$, $i, k = 1, ..., M_u$ gespeichert werden. Mit der Abbildung Φ werden offensichtlich die statischen Daten (siehe Abschnitt 5.1) von den dynamischen gitterabhängigen Daten (abhängig von Φ) getrennt. Dadurch erreicht man, dass die lokalen Matrizen nur einmal berechnet und gespeichert werden müssen.

Diffusionsterm

Der Diffusionsterm enthält Gradienten, die mit der Kettenregel (5.3) transformiert werden:

$$\begin{aligned}
&\int_K \nabla\varphi_i\nabla\varphi_k d\boldsymbol{x} = \det(\Phi)\int_{K_0}(\nabla\hat{\varphi}_i)^T\Phi^{-1}\left(\nabla\hat{\varphi}_k\Phi^{-1}\right)^T d\boldsymbol{\xi} \\
&= \det(\Phi)\int_{K_0}(\nabla\hat{\varphi}_i)^T\left(\Phi\Phi^T\right)^{-1}\nabla\hat{\varphi}_k d\boldsymbol{\xi} = \det(\Phi)\sum_{j=1}^{d}\sum_{l=1}^{d} b_{jl}\int_{K_0}\frac{\partial\hat{\varphi}_i}{\partial\xi_j}\frac{\partial\hat{\varphi}_k}{\partial\xi_l}d\boldsymbol{\xi}, \\
&\text{wobei } \left(\Phi\Phi^T\right)^{-1} = \{b_{jl}\}, \quad b_{jl} = b_{lj}.
\end{aligned} \tag{5.6}$$

Diese Trennung von statischen und dynamischen Daten liefert $\frac{d(d+1)}{2}$ unterschiedlicher symmetrischer lokaler Matrizen durch die Symmetrieeigenschaft der Matrix

$\left(\Phi\Phi^T\right)^{-1}$:

$$L_{jl\,(j\leq l)} = \{a_{ik}\}, \quad \begin{cases} a_{ik} = \displaystyle\int_{K_0} \frac{\partial\varphi_i}{\partial\xi_j}\frac{\partial\varphi_k}{\partial\xi_l} d\boldsymbol{\xi} & \text{falls } j = l, \\ a_{ik} = \displaystyle\int_{K_0} \left(\frac{\partial\varphi_i}{\partial\xi_j}\frac{\partial\varphi_k}{\partial\xi_l} + \frac{\partial\varphi_i}{\partial\xi_l}\frac{\partial\varphi_k}{\partial\xi_j}\right) d\boldsymbol{\xi} & \text{falls } j \neq l. \end{cases} \tag{5.7}$$

Druckterm

Sei $\Phi^{-1} = \{c_{gl}\}$, dann gilt für den Druckterm:

$$\int_K \psi_i \frac{\partial\varphi_k}{\partial x_l} d\boldsymbol{x} = \det(\Phi) \sum_{g=1}^{d} c_{gl} \int_{K_0} \hat{\psi}_i \frac{\partial\hat{\varphi}_k}{\partial\xi_g} d\boldsymbol{\xi}, \tag{5.8}$$

wobei durch partielle Ableitung d verschiedene lokale Matrizen $(M_u \times M_p)$ entstehen.

Konvektionsterm

Der nichtlineare Konvektionsterm enthält ein Produkt mit drei Ansatzfunktionen und wird folgendermaßen transformiert:

$$\begin{aligned} &\int_K \varphi_i \frac{\partial\varphi_j}{\partial x_l}\varphi_k d\boldsymbol{x} = \det(\Phi) \int_{K_0} \hat{\varphi}_i \left(\nabla\hat{\varphi}_j \cdot \Phi_l^{-1}\right) \hat{\varphi}_k d\boldsymbol{\xi} \\ &= \det(\Phi) \sum_{g=1}^{d} c_{gl} \int_{K_0} \hat{\varphi}_i \frac{\partial\hat{\varphi}_j}{\partial\xi_g}\hat{\varphi}_k d\boldsymbol{\xi}. \end{aligned} \tag{5.9}$$

Das erzeugt d dreidimensionale oder $d * M_u$ zweidimensionale symmetrische lokale Matrizen:

$$L_{gj} = \{a_{ik}\}, \qquad a_{ik} = \int_{K_0} \hat{\varphi}_i \frac{\partial\hat{\varphi}_j}{\partial\xi_g}\hat{\varphi}_k d\boldsymbol{\xi}, \qquad j = 1, ..., M_u, \quad g = 1, ..., d. \tag{5.10}$$

Kontinuitätsgleichung

Die GFEM-Diskretisierung der Kontinuitätsgleichung mit u_l als Komponenten des Geschwindigkeitsvektors ist

$$\sum_{l=1}^{d} \left(\sum_{i=1}^{M_u} \int_K \psi_k \frac{\partial\varphi_i}{\partial x_l} d\boldsymbol{x} \right) u_l = 0. \tag{5.11}$$

Mit demselben Transformationsverfahren bekommt man:

$$\det(\Phi) \sum_{l=1}^{d} \left(\sum_{g=1}^{d} c_{lg} \right) \left(\sum_{i=1}^{M_u} \int_{K_0} \hat{\psi}_k \frac{\partial\hat{\varphi}_i}{\partial\xi_l} d\boldsymbol{\xi} \right) u_l = 0. \tag{5.12}$$

Das liefert d lokale Matrizen $(M_p \times M_u)$, die genau die transponierten lokalen Matrizen des Druckterms sind.

Für die rechte Seite der Gleichung (5.4) kann auch ein lokaler Vektor mit den Werten $\int_{K_0} \hat{\varphi}_k d\boldsymbol{\xi}$ gespeichert werden.

Zu jedem Glied eines beliebigen Einschritt-Schema der Navier-Stokes-Gleichungen gibt es also eine lokale Matrix, die eine GFEM verwirklicht. Ein Term der Navier-Stokes-Gleichungen wird oft durch mehrere Summanden in einem Schema diskretisiert, wie z.B. immer der Zeitterm. Dadurch werden die gleichen lokalen Matrizen mehrfach gebraucht. Deshalb braucht man im Modell einer diskreten Gleichung (Abb. 5.6) sowohl Einzelterme, die eindeutig die notwendigen lokalen Matrize bestimmen, als auch das ganze Schema selbst, das im Assemblierungsprozess (Abschnitt 5.2.4) benutzt wird.

Merkmaldiagramme und Zusammenfassung

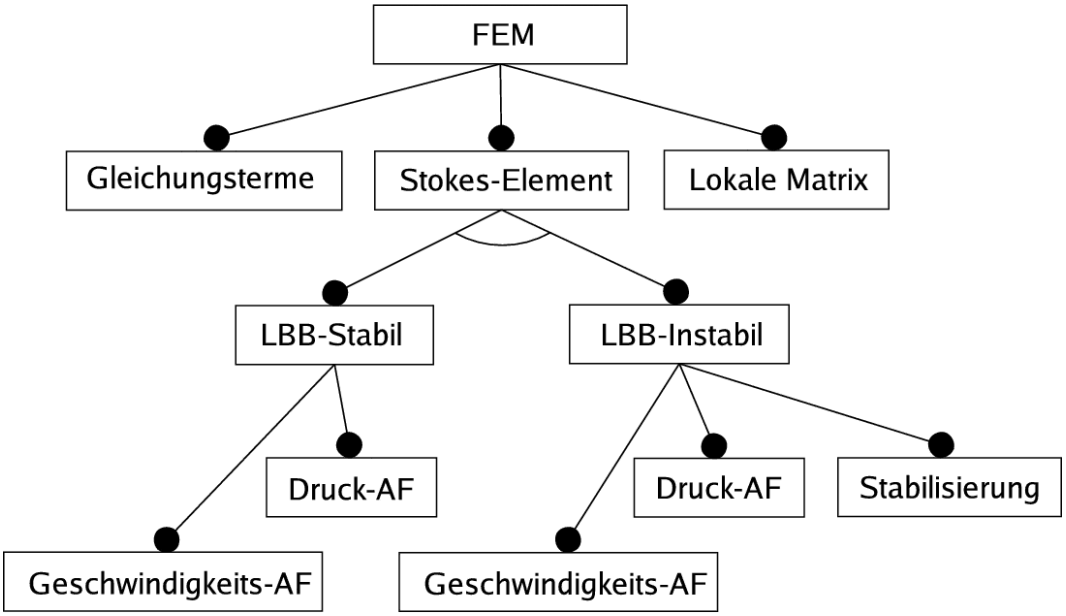


Abbildung 5.7: Merkmaldiagramm des Ansatzfunktionen-Konzepts

Das Konzept **FEM** ist nicht vom Gitter selbst, sondern von seinem Typ abhängig, weil die approximierenden Ansatzfunktionen (AF) auf Referenz-Elementen (Einheits-Dreieck, -Quadrat, etc.) definiert werden. Für die Navier-Stokes-Gleichungen müssen gemischte Stokes-Elemente benutzt werden, die aus zwei AF-Definitionen bestehen. Wenn aber ein LBB-instabiles Stokes-Element (Abschnitt 3.4.3) gewählt wird, muss auch ein Stabilisierungsverfahren angewendet werden (Abb. 5.7). Die Stabilisierung erfolgt durch zusätzliche Terme in der rechten Seite der Kontinuitätsgleichung, deshalb gehört dieses Merkmal auch zum Konzept **Gleichung**. Mit einem bestimmten Stokes-Element für einen dementsprechend diskretisierten o.g. Gleichungsterm wird eine lokale Steifigkeitsmatrix im Rahmen des FEM-Konzepts erzeugt.

Eine AF-Definition beinhaltet die Aufgaben-Dimension, den Geometrie-Typ und die polynomiale Ordnung, falls die AF konform ist. Im Gegenfall muss die AF eine besondere Definition bekommen. Nichtkonforme Ansatzfunktionen werden häufig erfolgreich eingesetzt, wie z.B. das Crouzeix-Raviart-Element [26, 7] oder das Rannacher-Turek-Element [140, 142, 151], lassen sich aber nicht wie konforme Elemen-

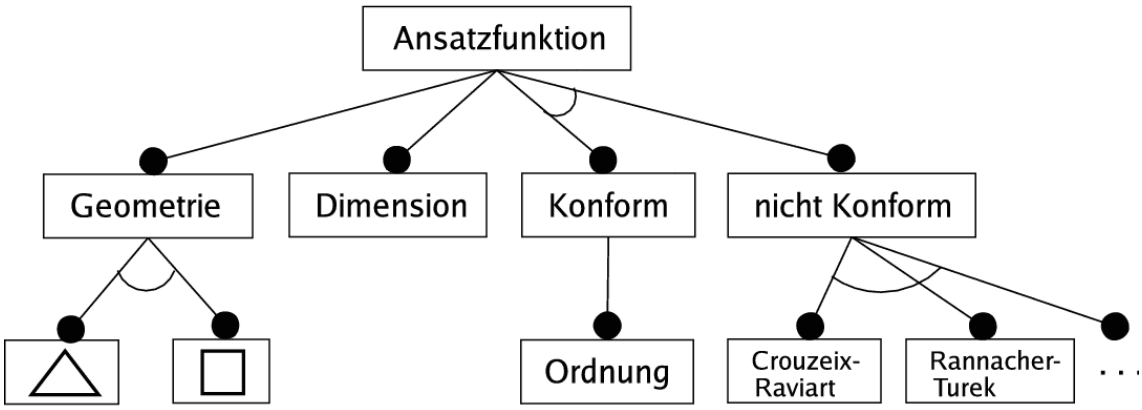


Abbildung 5.8: Merkmaldiagramm des Stokes-Element-Konzepts

te (Abschnitt 4.3.1) algorithmisch zuordnen. Der Geometrie-Typ bezeichnet ein Dreieck- und Tetraeder-Gitter oder ein Viereck- und Hexaeder-Gitter zusammen mit der Dimension (Abb. 5.8). Einige andere selten benutzte 3D-Gitter bestehend z.B. aus Prismen oder Pyramiden können normalerweise durch einfache Transformationen auf Tetraeder-Gitter zurückgeführt werden. Beliebige FVM- und FDM-Raumapproximationen können auch in der Form von lokalen Matrizen dargestellt werden.

5.2.4 Assemblierung

Das Matrix-Assemblierungs-Konzept verbindet die Konzepte **Gitter** und **Gleichung** und produziert ein lineares Gleichungssystem (5.13) mit Hilfe des FEM-Konzepts.

$$A\boldsymbol{x} = \boldsymbol{b}. \tag{5.13}$$

Nun wird ein reelles Gitter betrachtet. Die Freiheitsgrade dieses Gitters sind global durchnummeriert. Der Assemblierungs-Prozess muss alle Freiheitsgrade des Gitters durchlaufen. Generell gibt es zwei Hauptstrategien dafür: Kantenweise und zellenweise. Die erste Strategie zerlegt eine lokale Matrix auf Kanteneinträge und ist bekannt effizienter zu sein [35, 106, 107, 157], wird aber normalerweise in der Festkörpersimulationen eingesetzt, wo ausschließlich Approximationen niedriger Ordnung mit auf den Kanten liegenden Freiheitsgraden gewählt werden. In einer Strömungssimulation werden dagegen häufig quadratische und Approximationen höherer Ordnung mit den innerhalb der Zelle liegenden Freiheitsgraden benutzt (Abschnitt 3.4). Deshalb wird die zweite Strategie universell eingesetzt. Für jedes Gitter-Element und Gleichungsglied wird die zugehörige lokale Matrix genommen (Abschnitt 5.2.3). Jede Zeile und Spalte dieser Matrix bekommt einen Index aus der globalen Freiheitsgradnummerierung. Diese Matrix wird als eine schwachbesetzte Matrix zur globalen Steifigkeitsmatrix A addiert, wobei die Indizes die Stellen der Einträge in der globalen Matrix A bestimmen (Abb. 5.9). In der diskreten Gleichung können auch Glieder auftreten, die Unbekannte aus einem vorhergehenden Zeitschritt oder einen bekannten Vektor enthalten. Ein solcher Vektor wird mit der zugehörigen lokalen Matrix wie mit einer schwachbesetzten Matrix gleicher Dimension multipliziert und damit werden Beiträge zur rechten Seite $\boldsymbol{b}$ entsprechend der Zeilenindizes der lokalen Matrix erzeugt.

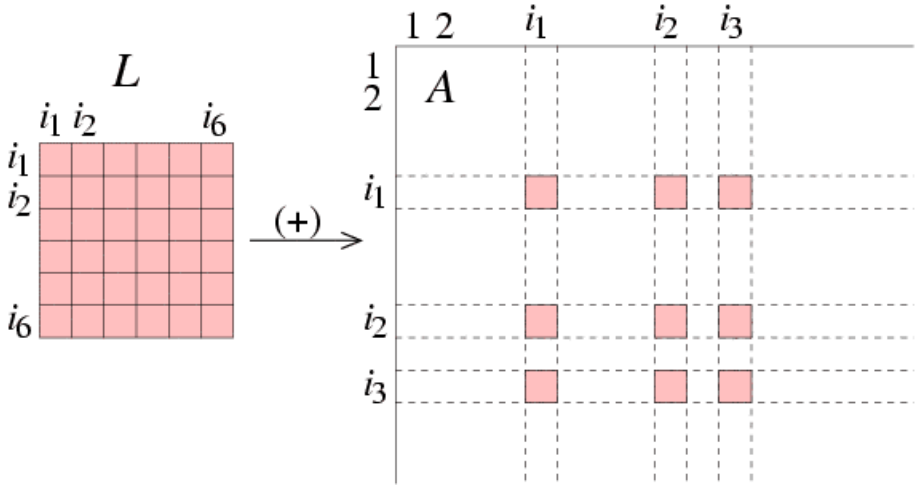


Abbildung 5.9: Lokale und globale Steifigkeitsmatrix im Assemblierungsprozess

Eine auf diese Weise assemblierte Matrix A ist immer singulär. Es müssen noch Randbedingungen berücksichtigt werden. Dirichlet-Randbedingungen $\boldsymbol{x}_D$ geben bestimmte Werte für den unbekannten Vektor $\boldsymbol{x}$ auf einem Rand Γ_D vor. Dann kann man das lineare Gleichungssystem (5.13) blockweise aufschreiben:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{12}^T & A_{22} \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{x}_D \end{pmatrix} = \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{b}_D \end{pmatrix} \tag{5.14}$$

Die zweite Zeile in 5.14 ist überflüssig, da sie nur bekannte Daten enthält. Die erste Zeile transformiert sich zu

$$A_{11}\boldsymbol{x} = \boldsymbol{b} - A_{12}\boldsymbol{x}_D \qquad \text{oder} \qquad \tilde{A}\boldsymbol{x} = \tilde{\boldsymbol{b}}. \tag{5.15}$$

Die Randpunkte mit definierten Neumann-Randbedingungen werden genau so, wie die inneren Punkte, in die Steifigkeitsmatrix $\tilde{A}$ assembliert, da nur die Normal-Ableitung ($\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{n}} = \boldsymbol{g}$) auf dem Randstück Γ_N und keine Geschwindigkeit selbst in den Randpunkten vorgegeben ist. Für diese Punkte muss das Randintegral $\int_{\Gamma_N} g d\boldsymbol{x}$ in die rechte Seite der entsprechenden Gleichung als eine zusätzliche äußere Kraft hinzugefügt werden.

Komplexität

Wir wollen nun die Komplexität des auf den lokalen Matrizen (Abschnitt 5.2.3) basierten Assemblierungsprozesses abschätzen. Sie besteht für jede Gitterzelle aus den folgenden Schritten:

1. Berechnung von $\det(\Phi)$, Φ^{-1} und $(\Phi\Phi^T)^{-1}$ der linearen Transformation Φ;
2. Anwendung der linearen Transformation Φ auf die statischen lokalen Matrizen bzw. Berechnung von aktuellen lokalen Matrizen durch die Formeln (5.5),(5.6),(5.8),(5.9),(5.11);
3. Addition der aktuellen lokalen Matrizen in die globale Steifigkeitsmatrix.

Die Tabelle 5.3 stellt die Komplexität für jedes Gleichungsglied nach diesen Schritten dar. Dabei ist die Symmetrie der lokalen Zeitterm-, Diffusions- und Konvektions-Matrizen so berücksichtigt, dass der Speicherplatz und die Komplexität von Matrix-Konstanten-Operationen nur $\frac{1}{2}M_u(M_u+1)$ statt M_u^2 betragen. Bei der Addition der lokalen Matrizen (3. Schritt) bringt aber die Symmetrie keine Vorteile. In den Komplexitäts-Abschätzungen wurden sowohl Multiplikations- als auch Additions-Operationen mitgezählt, da sie etwa gleichen Rechenaufwand haben (siehe Anhang A.3). Deshalb ist die Komplexität eines Matrix-Vektor-Produktes und eines Skalar-Produktes gleich $2M_u^2$ und $2M_u$ entsprechend.

		Komplexität	**RAM**
1	$\det(\Phi)$	$2d^2-2d$	1
	Φ^{-1}	d^d+d^2	d^2
	$(\Phi\Phi^T)^{-1}$	$d^d+d^3+d^2$	$\frac{1}{2}d(d+1)$
2	Zeitterm	$\frac{1}{2}M_u(M_u+1)$	$\frac{1}{2}M_u(M_u+1)$
	Diffusionsterm	$\frac{1}{2}(d^2+d-1)M_u(M_u+1)$	$\frac{1}{4}d(d+1)M_u(M_u+1)$
	Druckterm	$d(2d-1)M_uM_p$	dM_uM_p
	Konvektionsterm	$\frac{1}{2}d(2d-1)M_u^2(M_u+1)$	$\frac{1}{2}dM_u^2(M_u+1)$
	Kontinuität	$d(2d-1)M_uM_p$	dM_uM_p
3	Zeitterm	M_u^2	
	Diffusionsterm	$\frac{1}{2}d(d+1)M_u^2$	
	Druckterm	dM_uM_p	
	Konvektion (i)	$2dM_u^3+(2d-1)M_u^2$	
	Konvektion (j)	$2d^2M_u^3+d^2M_u^2$	
	Kontinuität	dM_uM_p	

Tabelle 5.3: Komplexität und Speicherverbrauch der Assemblierungsschritte

Eine besondere Aufmerksamkeit benötigt der nichtlineare Konvektionsterm, der d Summanden der Gleichung beinhaltet. Die Konvektion kann nur dann zur globalen Matrix addiert werden, wenn einen der beiden Geschwindigkeitsvektoren nach der Linearisierung aus dem vorhergehenden Zeitschritt bekannt ist. Dementsprechend hat die Konvektion im dritten Schritt die zwei Zeilen in der Tabelle 5.3, wenn der Vektor $\boldsymbol{u}_i$ oder der Vektor $\boldsymbol{u}_j$ bekannt ist. Die jeweiligen lokalen Matrizen bezeichnen wir $L_l(u_i), L_l(v_i)$ und $L_l(u_j), L_l(v_j)$, $l=\overline{1,d}$. Im ersten Fall hat die Addition eine deutlich kleinere Komplexität, weil sich die unbekannte Geschwindigkeit $\boldsymbol{u}_j$ ausklammern lässt und in den Klammern die gleiche Matrix bleibt:

$$\begin{aligned} &L_1(u_i)u_j+L_2(v_i)u_j=[L_1(u_i)+L_2(v_i)]u_j,\\ &L_1(u_i)v_j+L_2(v_i)v_j=[L_1(u_i)+L_2(v_i)]v_j, \end{aligned} \qquad L_l(u_i)=\{a_{jk}\}=\int_K \varphi_i\frac{\partial\varphi_j}{\partial x_l}\varphi_k d\boldsymbol{x}\,u_i \tag{5.16}$$

Das erklärt auch die Vorzüge des semi-expliziten Euler-Schemas. Der zweite und der dritte Assemblierungsschritt können für die Konvektion zusammengesetzt werden, um die gesamte Komplexität für die beiden Fälle zu reduzieren. Die Beziehung (5.16)

transformiert sich mit Hilfe der Formel (5.9) zu

$$
\begin{aligned}
&[c_{11}\hat{L}_1(u_i) + c_{21}\hat{L}_2(u_i)]u_j + [c_{12}\hat{L}_1(v_i) + c_{22}\hat{L}_2(v_i)]u_j \\
&\qquad = [\hat{L}_1(c_{11}u_i + c_{12}v_i) + \hat{L}_2(c_{21}u_i + c_{22}v_i)]u_j, \\
&[c_{11}\hat{L}_1(u_i) + c_{21}\hat{L}_2(u_i)]v_j + [c_{12}\hat{L}_1(v_i) + c_{22}\hat{L}_2(v_i)]v_j \\
&\qquad = [\hat{L}_1(c_{11}u_i + c_{12}v_i) + \hat{L}_2(c_{21}u_i + c_{22}v_i)]v_j,
\end{aligned}
\qquad \hat{L}_l(u_i) = \int_{K_0} \hat{\varphi}_i \frac{\partial \hat{\varphi}_j}{\partial \xi_l} \hat{\varphi}_k d\boldsymbol{\xi}\, u_i
$$

wobei zuerst die Vektoren in den runden Klammern und danach die Matrix-Vektor-Produkte berechnet werden. Wenn der Vektor $\boldsymbol{u}_j$ bekannt ist, können die Beziehungen nicht in ähnlicher Weise transformiert, sondern müssen direkt in der folgenden Form ausgewertet werden:

$$
\begin{aligned}
[c_{11}\hat{L}_1(u_j) + c_{21}\hat{L}_2(u_j)]u_i \quad &+ \quad [c_{12}\hat{L}_1(u_j) + c_{22}\hat{L}_2(u_j)]v_i \\
[c_{11}\hat{L}_1(v_j) + c_{21}\hat{L}_2(v_j)]u_i \quad &+ \quad [c_{12}\hat{L}_1(v_j) + c_{22}\hat{L}_2(v_j)]v_i.
\end{aligned}
$$

Dann ist die neue gesamte Komplexität des zweiten und des dritten Schrittes:

Konvektion (i): $2dM_u^3 + (2d-1)M_u^2 + d(2d-1)M_u$;

Konvektion (j): $2d^2M_u^3 + 2d^3M_u^2$.

Die erste Abschätzung ist deutlich kleiner als

$$
2dM_u^3 + (2d-1)M_u^2 + \frac{1}{2}d(2d-1)M_u^2(M_u+1)
$$

wegen dem letzten Summanden. Die zweite Abschätzung muss genauer untersucht werden, liefert aber dieselbe Schlussfolgerung für $d = 2,\ m \geq 1$ und $d = 3,\ m \geq 2$, was den üblichen praktischen Werten entspricht.

Strategien

Nach der Komplexitäts-Abschätzung für die Assemblierung einer Zelle betrachten wir den gesamten Assemblierungsprozess über alle N_T Zellen und die Zeitschritte. Die Größe M_S der globalen Steifigkeitsmatrix $\tilde{A}$ kann vor der Assemblierung aus dem Gitter exakt gewonnen werden, um die aufwendigen Speicherverwaltungsoperationen während der Assemblierung zu vermeiden. Die Matrix $\tilde{A}$ hat $N_u + N_p - N_D$ Zeilen, wobei N_D die Anzahl von Dirichlet-Randpunkten bezeichnet. Die Anzahl der nicht Null Einträge jeder Matrix-Zeile hängt von der Nachbarnzahl des entsprechenden Freiheitsgrads und der Gleichung (Impuls- oder Kontinuitätsgleichung) ab. Die Matrix $\tilde{A}$ benötigt einen großen Anteil des insgesamt vom Softwaresystem gebrauchten Speichers und kann für große Probleme schnell die Speichergrenze erreichen. In jeder Zeititeration wird eine neue Lösung $(\boldsymbol{u}, p)$ berechnet, deshalb muss die nichtlineare Konvektion neu assembliert werden. Die restlichen Einträge der Matrix $\tilde{A}$ ändern sich bei einem konstanten Gitter in der Zeit nicht. Diese Einträge könnten zur Optimierung des Assemblierungsprozesses zusätzlich gespeichert werden, wenn der Speicherplatz ausreichen würde. Diese Überlegungen führen zur folgenden Klassifizierung von Assemblierungsstrategien nach Speicherverbrauch und notwendiger

CPU-Zeit. Mit der Bezeichnung

$$K_\Phi = 2d^d + d^3 + 4d^2 - 2d$$

der gesamten Komplexität der Berechnung der Transformationsdaten (Tab. 5.3, 1. Zeile) erhält man:

I Große Probleme: Der Speicher reicht für zwei globale Steifigkeitsmatrizen nicht aus, deshalb ist eine vollständige Assemblierung der globalen Matrix in jeder Zeititeration erforderlich.

Speicherverbrauch: M_S.

Komplexität pro Zeitschritt:

$$N_T(2dM_u^3+\left(\frac{d^2}{2}+\frac{5d}{2}-1\right)M_u^2+2d(2d-1)M_uM_p+\left(\frac{5d^2}{2}+\frac{d}{2}-1\right)M_u+K_\Phi).$$

II Mittelgroße Probleme: Zwei Steifigkeitsmatrizen können gespeichert werden, deshalb werden alle Terme außer der Konvektion in eine zusätzliche Matrix assembliert. Die endgültige globale Matrix ist dann die Summe der zusätzlichen Matrix und der Konvektions-Assemblierung in jeder Zeititeration. Die wesentlich reduzierten Rechenkosten benötigen dabei den doppelten Speicherplatz für die Steifigkeitsmatrix.

Speicherverbrauch: $2M_S$.

Komplexität pro Zeitschritt: $N_T(2dM_u^3 + (2d-1)M_u^2 + d(2d-1)M_u + K_\Phi)$.

III Kleine Probleme: Der Speicher reicht für eine Steifigkeitsmatrix im vollbesetzten Format. Die globale Matrix wird als eine vollbesetzte Matrix assembliert und danach in die schwachbesetzte umgewandelt. Da die lokalen Matrizen als schwachbesetzte zu der globalen Matrix addiert werden, gewinnt man viel Rechenzeit bei der direkten Addition einer schwachbesetzten mit einer vollbesetzten Matrix. Der Gewinn ist in der Komplexität-Abschätzung nicht widergespiegelt, weil sie von der schwachbesetzten Matrix-Datenstruktur abhängt.

Speicherverbrauch: $(N_u + N_p - N_D)^2 + M_S$.

Komplexität pro Zeitschritt: $N_T(2dM_u^3 + (2d-1)M_u^2 + d(2d-1)M_u + K_\Phi)$.

- Zu jeder genannten Strategie gibt es die Möglichkeit, die Transformationsmatrizen Φ^{-1} und $(\Phi\Phi^T)^{-1}$ sowie Determinanten $\det(\Phi)$ für sämtliche Elemente vorher abzuspeichern.

  Zusätzlicher Speicherverbrauch: $N_T\left(\frac{3d^2}{2}+\frac{d}{2}+1\right)$.

  Komplexitätreduktion pro Zeitschritt: K_Φ.

Alle Komplexitäts-Abschätzungen wurden für das semi-explizite Euler-Schema (5.4) durchgeführt, obwohl dieselben Abschätzungen für andere Schemata mit Hilfe der Tabelle 5.3 leicht möglich sind.

Die Assemblierungs-Strategie kann auch als ein Domänenmodell-Parameter angenommen werden, muss aber erst in der Laufzeit nach der Abschätzung des verfügbaren Speicherplatzes gewählt werden.

5.2.5 Gleichungssystemlöser

Für die diskretisierten und linearisierten inkompressiblen Navier-Stokes-Gleichungen hat das lineare Gleichungssystem (5.15) folgende Blockstruktur:

$$\begin{pmatrix} F & B^T \\ B & -\beta C \end{pmatrix} \begin{pmatrix} \boldsymbol{u} \\ p \end{pmatrix} = \begin{pmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \end{pmatrix}, \tag{5.17}$$

wobei der Ausdruck $-\beta C$ stabilisierte GFEM-Approximationen beschreibt. Der Fall $\beta = 0$ entspricht den im Abschnitt (3.5) betrachteten stabilen Stokes-Elementen. Die Matrix F ist der diskrete Konvektions-Diffusions-Operator und die Matrix B ist der negative diskrete Divergenz-Operator, der die Geschwindigkeits-Druck-Kopplung beinhaltet. Alle Blockmatrizen in (5.17) sind schwach besetzt, d.h. jede Zeile bzw. Spalte der Matrix enthält nur wenige von Null verschiedene Elemente, deren Anzahl unabhängig von der Matrix-Dimension ist.

Zur Lösung des linearen Gleichungssystems (5.17) werden an erster Stelle iterative Verfahren betrachtet [13, 69, 76, 90, 111, 145], da direkte Löser einen großen zusätzlichen Speicher-Platz benötigen und deshalb für große Strömungsprobleme i.a. nicht anwendbar sind. Die Matrix F ist unsymmetrisch und indefinit. Das legt die Wahl der iterativen Krylov-Unterraum-Verfahren nahe (Abb. 5.10). Die Tabelle 5.4 fasst die Verfahren mit dem notwendigen Speicherplatz für temporäre Vektoren und die Matrix-Vektor-Produkte-, Skalar-Produkte-, und Vektor-Konstanten-Operationen-Anzahl pro Iteration zusammen. Die Verfahren teilen sich nach dem Grundprinzip in die zwei Klassen:

- Minimierung der Residuumsnorm: MINRES, GMRES, QMR, TFQMR;
- Lösungssuche durch konjugierte Gradienten: CG, CGS, BiCGStab, QMRBCGStab.

Die CG- und MINRES-Verfahren sind nur für symmetrische Systeme geeignet, die z.B. bei Stokes-Problemen vorkommen; sie erfordern deshalb wesentlich kleineren Rechenaufwand.

Das GMRES-Verfahren ist der einzige Solver für nichtsymmetrische Systeme, der nur ein Matrix-Vektor-Produkt pro Iteration ausführt, benötigt aber eine von der Iterationsanzahl abhängige Menge von Vektoren. Diese Vektoren sind eine Orthonormalbasis zur Berechnung der nächsten Näherungslösung in jeder Iteration, deshalb wird das Verfahren mit jeder Iteration langsamer. Es zeichnet sich durch ein monoton absteigendes Konvergenzverhalten aus, wobei man die Iterationsanzahl durch günstige Präkonditionierung (Abschnitt 5.2.6) reduzieren kann. Bei einer Präkonditionierung ist die Modifikation FGMRES besonders günstig: Durch Speicherung eines zusätzlichen Vektors pro Iteration führt das Verfahren nur eine Anwendung des Präkonditionierers durch.

Das QMR-Verfahren führt eine Multiplikation mit $\tilde{A}^T$ pro Iteration durch. Für eine schwachbesetzte Matrix ist es nicht immer einfach, die transponierte Matrix zu bekommen, wenn sie z.B. im CSR-Format [145] dargestellt ist. Deshalb existiert auch die transponierungsfreie Modifikation dieses Verfahrens, TFQMR.

In der Arbeit von Meister [110] wurden fünf Verfahren (GMRES, CGS, BiCGStab, TFQMR und QMRBCGStab) für zwei zweidimensionale Strömungsprobleme

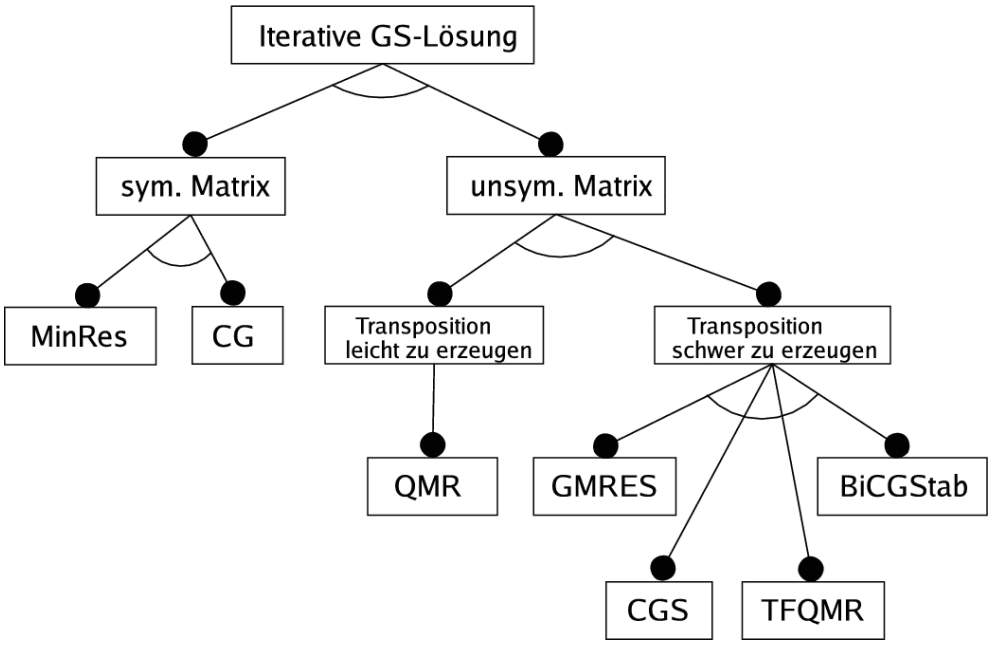


Abbildung 5.10: Merkmaldiagramm des iterativen Gleichungssystemlöser-Konzepts

Abkürzung	**Name**	**RAM**	**Komplexität**		
			MV	dot	V
CG	Conjugate Gradient [80]	4	1	2	3
MINRES	MINimal RESiduum [127]	4	1	2	7
GMRES	Generalized Minimal RESiduum [147]	$2k$+5	1	k	k
FGMRES	Flexible GMRES [144]	$3k$+5	1	k	k
QMR	Quasi-Minimal Residuum [56]	9	2	3	9
TFQMR	Transpose Free QMR [55]	9	2	4	10
CGS	Conjugate Gradient Squared [155]	7	2	2	7
BiCGStab	BiConjugate Gradient Stabilized [166]	7	2	4	6
QMRBCGStab	QMR + BiCGStab [30]	8	2	6	8

Tabelle 5.4: Iterative Löser für indefinite lineare Gleichungssysteme

mit unterschiedlichen Präkonditionierer verglichen. Mit dem ILU-Präkonditionierer wird das BiCGStab-Verfahren als der schnellste Löser empfohlen, das auch mit Ergebnissen in [31] übereinstimmt. Die Arbeit [181] zeigt dagegen eine schnellere GMRES-Lösung für eine FE-Diskretisierung und behauptet, dass der Unterschied noch von der Besetzung der Matrix abhängt. Wenn als Präkonditionierung die Skalierung der Matrix A genommen wird [110], zeigt das TFQMR-Verfahren dagegen die besten Ergebnisse. Das GMRES-Verfahren ist dabei stark vom gewählten Präkonditionierer abhängig.

Alle iterativen Verfahren sind stark von der Konditionszahl der System-Matrix abhängig. Da die Steifigkeitsmatrix gewöhnlich eine große Konditionszahl hat, versucht man die Matrix günstig zu präkonditionieren, um die Iterationsanzahl wesentlich zu reduzieren.

5.2.6 Präkonditionierung

Wir befassen uns hier nicht mit allgemeinen Präkonditionierungs-Verfahren, die in vielen Arbeiten [13, 23, 69, 90, 111, 143, 145, 146, 148] aufgeführt sind, sondern interessieren uns für die speziellen Block-Präkonditionierer, die die Matrix-Blockstruktur (5.17) ausnutzen. Für das Stokes-Problem mit einer symmetrischen Matrix F mit Hilfe der Eigenvektor-Analyse werden solche Präkonditionierer in der Blockdiagonalform

$$P = \begin{pmatrix} F & 0 \\ 0 & -X \end{pmatrix}, \qquad C_p = \int_\Omega \psi_i \psi_j d\boldsymbol{x} \tag{5.18}$$

gesucht, wobei $X = \mathrm{diag}(C_p)$, $X = C_p$ [184, 154] oder $X = \frac{1}{\nu} C_p$ [45] und C_p die Druck-Massen-Matrix ist. Das gilt auch für zeitabhängige Probleme [105].

Für das Navier-Stokes-Problem mit der nichtsymmetrischen Matrix F wird die Steifigkeitsmatrix für die stabilen Stokes-Elemente ($\beta = 0$) durch eine Block-LU-Faktorisierung zerlegt:

$$\begin{pmatrix} F & B^T \\ B & 0 \end{pmatrix} = \begin{pmatrix} I & 0 \\ BF^{-1} & I \end{pmatrix} \begin{pmatrix} F & B^T \\ 0 & -BF^{-1}B^T \end{pmatrix}. \tag{5.19}$$

Wenn man mit $S = BF^{-1}B^T$ den Schur-Komplement-Operator bezeichnet, ergibt sich

$$\begin{pmatrix} F & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} F & B^T \\ 0 & -S \end{pmatrix}^{-1} = \begin{pmatrix} I & 0 \\ BF^{-1} & I \end{pmatrix}. \tag{5.20}$$

Deshalb wird ein rechter Präkonditionierer in der Form

$$P = \begin{pmatrix} F & B^T \\ 0 & -X \end{pmatrix} \tag{5.21}$$

gewählt. Das mit P präkonditionierte GMRES-Verfahren mit $X = S$ benötigt genau zwei Iterationen bis zur exakten Lösung [117]. Da die exakte Berechnung von S aufwendig ist, kann die Matrix X unterschiedlich approximiert werden:

- $X_M = \frac{1}{\nu} C_p$ (vorgeschlagen in [45]);
- $X_B = (BB^T)(BFB^T)^{-1}(BB^T)$ (vorgeschlagen in [46, 79]);
- $X_R^{-1} = \frac{1}{2}(X_M^{-1} + X_B^{-1})$ (vorgeschlagen in [89]);
- $X_G = D_p F_p^{-1} C_p$ (vorgeschlagen in [89, 48]).

Die Matrizen D_p und F_p sind entsprechend die diskreten Diffusions- und Konvektions-Diffusions-Operatoren für den Druck. Die umfangreichen Tests und Untersuchungen aller vier Versionen [47, 49, 89, 102, 153, 183] zeigen die Unabhängigkeit der GMRES-Iterationszahl vom Raumdiskretisierungsparameter h, obwohl die Iterationsanzahl für kleinere Werte von ν bzw. größere Reynoldszahl wächst.

Für die Faktorisierung der inversen Matrix P in der Form (5.21) gilt

$$P^{-1} = \begin{pmatrix} F^{-1} & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} I & B^T \\ 0 & -I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & X^{-1} \end{pmatrix}. \tag{5.22}$$

Die Implementierung des Präkonditionierers schließt also die Berechnung von F^{-1} und X^{-1} ein.

- Für die Lösung eines diskreten Konvektions-Diffusions-Problems mit der Matrix F sind Multigrid- [139] oder Blockdiagonal-Präkonditionierer [101] gut geeignet. Die Matrix F ändert sich in jeder Zeititeration.
- Die Matrix X_M^{-1} kann mit dem klassischen CG-Verfahren und der einfachen diagonalen Präkonditionierung approximiert werden.
- Es gibt keinen einfachen Weg, die Matrix $(BB^T)^{-1}$ zur Bestimmung von X_B^{-1} zu berechnen. Die Auswertung von $(BB^T)^{-1}$ geschieht aber nur einmal bei zeitabhängigen Problemen.
- Zur Berechnung von X_G^{-1} außer der Matrix C_p^{-1} braucht man noch die Matrix D_p^{-1} mit dem allgemeinen Präkonditionierer, wie z.B. die Multigrid-Technik, zu bestimmen, wobei diese Matrix genau wie die Massenmatrix C_p noch zusätzlich assembliert werden muss.

Das bekannte SIMPLE-Verfahren [129] kann auch als ein Präkonditionierer dargestellt werden [97, 179, 180]:

$$P^{-1} = \begin{pmatrix} I & -D^{-1}B^T \\ 0 & I \end{pmatrix} \begin{pmatrix} F & 0 \\ B & -BD^{-1}B^T \end{pmatrix}^{-1}, \qquad D = \operatorname{diag}(F), \tag{5.23}$$

wobei die inversen Matrizen F^{-1} und B^{-1} approximiert werden müssen.

Man kann auch ähnlich zu (5.19) eine unvollständige Block-LU-Faktorisierung durchzuführen [64, 173]

$$\begin{aligned} \tilde{A} &= \begin{pmatrix} F & 0 \\ B & -BF^{-1}B^T \end{pmatrix} \begin{pmatrix} I & F^{-1}D^T \\ 0 & I \end{pmatrix} \\ &\approx \begin{pmatrix} F & 0 \\ B & -BHB^T \end{pmatrix} \begin{pmatrix} I & GD^T \\ 0 & I \end{pmatrix} = \begin{pmatrix} F & FGB^T \\ B & B(G-H)B^T \end{pmatrix}, \end{aligned} \tag{5.24}$$

wobei zur Approximation von $\tilde{A}$ die zwei Matrizen G und H eingeführt wurden. Wenn die Matrix C_u die Geschwindigkeits-Massenmatrix bezeichnet, dann entspricht die Wahl $G = H = \Delta t M_u^{-1}$ der algebraischen Version der Chorin-Temam-Projektionsmethode [130] (Abschnitt 3.1.2). Die Wahl $H = \Delta t M_u^{-1}$ und $G = F$ liefert die sog. Yosida-Methode [135, 136].

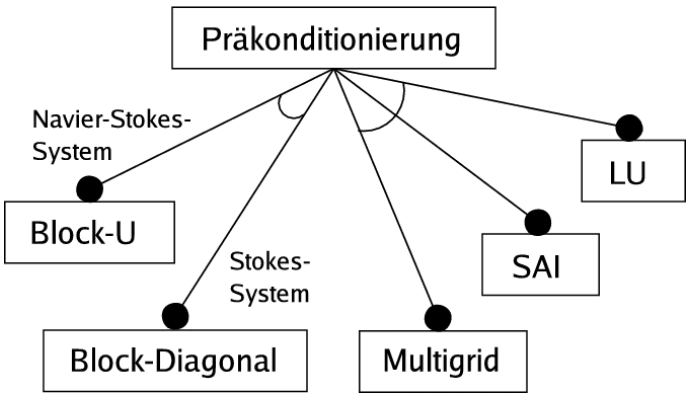


Abbildung 5.11: Merkmaldiagramm des Präkonditionierungs-Konzepts

Das Präkonditionierungs-Konzept enthält also die Block-Präkonditionierer (5.18) und (5.21) entsprechend für die Stokes- und die Navier-Stokes-Gleichungen. Außerdem müssen noch allgemeine Präkonditionierer der Multigrid-, LU- und SAI-Klassen (engl. sparse approximate inverse) zur Berechnung von F^{-1} und X^{-1} implementiert werden.

5.2.7 Parallelisierung

Der Parallelisierungsprozess erfolgt in drei Etappen:

1. Partitionierung des Gitters auf überlappende oder nichtüberlappende (Abschnitt 5.2.1) Gleichgewichtsteile mit möglichst kleinerem gemeinsamen Rand zwischen jeweils zwei Teilen. Die überlappende Partitionierung führt zu zusätzlichem Kommunikations- und Rechen-Aufwand wegen der mehrfach gespeicherten Randzellen in den Gitterpartitionen und wird nur im Fall eines Upwindverfahrens benutzt, das Freiheitsgrade aus den benachbarten Zellen mit einbezieht;
2. Parallele Assemblierung der Steifigkeitsmatrix A;
3. Parallele Lösung des linearen Gleichungssystems.

In diesem Abschnitt werden die zweite und die dritte Etappe behandelt, wobei die Partitionierung mit einem externen Graphenpartitionierer durchgeführt werden kann (Abschnitt 5.2.1).

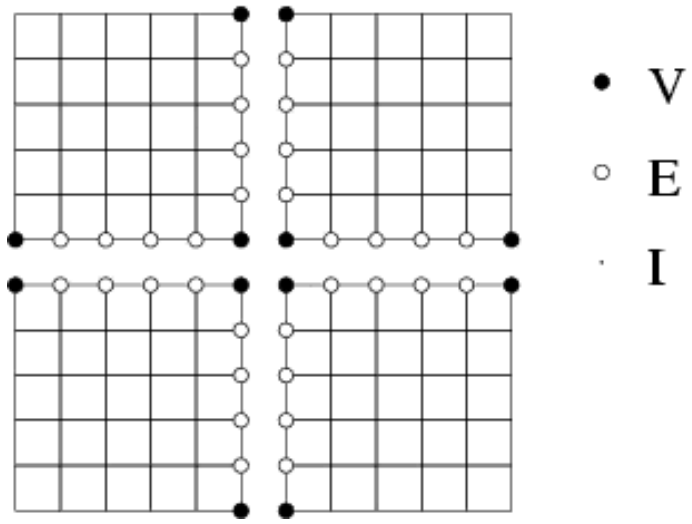


Abbildung 5.12: Punktebezeichnung eines partitionierten Viereckgitters

Wir betrachten die Parallelisierung auf dem algebraischen Niveau, weil die verteilten Stokes-/Oseen-Probleme auf die äquivalenten Lineare-Algebra-Probleme zurückgeführt werden können. Die Grundlagen der Parallelisierung werden nach der Arbeit von Haase [76] dargestellt. Sei das Dreieck- oder Viereckgitter $\mathcal{T}$ in die Teilgitter $\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_P$ zerlegt (z.B. Abb. 5.12 mit $P = 4$ auf einem Viereckgitter in zwei Dimensionen). Wir unterscheiden drei Arten von Knoten durch die folgenden Indizes:

(I) Punkte bzw. Freiheitsgrade im Teilgebietsinneren ($\mathcal{N}_I$),

(E) Punkte im Inneren der Teilgebietskanten ($\mathcal{N}_E$),

(V) Punkte am Beginn oder am Ende einer Teilgebietskante ($\mathcal{N}_V$).

Die Gesamtzahl der Punkte ergibt sich zu $N = \mathcal{N}_V + \mathcal{N}_E + \mathcal{N}_I$. Zur Vereinfachung der Darstellung werden zuerst V-, dann E- und I-Punkten nummeriert. Die Knotenzuordnung in den Teilgebieten wird mit Hilfe der Koinzidenzmatrizen $K_i\,(i = 1, ..., P)$ symbolisch repräsentiert. Die Matrix $K_i\,(N_i \times N)$ ist eine Boolesche Matrix, welche einen globalen Vektor $\boldsymbol{u}$ auf den lokalen Vektor $\boldsymbol{u}_i$ abbildet. Dann erscheinen die Einträge für innere Punkte genau einmal pro Zeile und Spalte in den Matrizen K_i. Die Einträge der Koppelknoten treten so oft in K_i auf, wie es Teilgebiete gibt, zu denen sie gehören. Die Anzahl der Einträge enthält die Diagonalmatrix

$$R = \sum_{i=1}^{P} K_i^T K_i. \tag{5.25}$$

Wir definieren nun zwei Arten von Vektoren auf den verteilten Teilgittern, wobei die Vektorelemente die bestimmten Werte in den Gitterpunkten speichern.

Definition 5.1. *(**Akkumulierter Vektor**) Der Vektor $\underline{\boldsymbol{u}}$ wird in Prozessor $\mathbb{P}_i$ als*

$$\underline{\boldsymbol{u}}_i = K_i \underline{\boldsymbol{u}}$$

gespeichert, d.h. der Vektor $\underline{\boldsymbol{u}}_i$ besitzt die vollen Werte in den zum Teilgitter $\mathcal{T}_i$ gehörenden Punkten.

Definition 5.2. *(**Verteilter Vektor**) Der Vektor $\boldsymbol{b}$ wird in Prozessor $\mathbb{P}_i$ als $\boldsymbol{b}_i$ so gespeichert, dass*

$$\boldsymbol{b} = \sum_{i=1}^{P} K_i^T \boldsymbol{b}_i$$

gilt, d.h. die Vektorelemente b_V und b_E besitzen nur einen Teil des vollen Wertes und erst ihre globale Akkumulation erzeugt den vollen Wert.

Die globale Steifigkeitsmatrix A wird im verteilten Sinne analog zum verteilten Vektor abgespeichert und daher als verteilte Matrix klassifiziert.

$$A = \sum_{i=1}^{P} K_i^T A_i K_i, \tag{5.26}$$

wobei A_i die zum Teilgitter $\mathcal{T}_i$ gehörende Steifigkeitsmatrix darstellt. Die verteilten Matrizen A_i können deswegen ohne Kommunikation auf jedem Prozessor $\mathbb{P}_i$ parallel assembliert und gespeichert werden. So haben wir die zweite Etappe des Parallelisierungsprozesses geklärt.

Operationen mit Kommunikation

Die Vektor-Operationen mit Vektoren gleicher Art können offensichtlich ohne Kommunikation ausgeführt werden. Die Umwandlung eines verteilten in einen akkumulierten Vektor erfordert Kommunikation:

$$\underline{\boldsymbol{b}}_i = K_i \sum_{i=1}^{P} K_i^T \boldsymbol{b}_i, \tag{5.27}$$

wobei die Randpunktwerte $\boldsymbol{b}_V$ und $\boldsymbol{b}_E$ zwischen jeweils zwei Teilgittern ausgetauscht werden müssen. Die andere Umwandlung eines akkumulierten in einen verteilten Vektor ist nicht eindeutig. Eine Approximationsmöglichkeit besteht in der lokalen Division jeder Vektorkomponente durch die Anzahl der anliegenden Teilgebiete, z.B.

$$\boldsymbol{b}_i = R^{-1} \underline{\boldsymbol{b}}_i. \tag{5.28}$$

Die Matrix R aus (5.25) kann für jeden Prozessor gespeichert werden und wird nur dann mit der zusätzlichen Kommunikation aktualisiert, wenn das Gitter geändert wird, z.B. bei der Adaption.

Das Skalarprodukt von Vektoren verschiedener Vektortypen erfordert bzgl. der Kommunikation nur die Summation einer reellen Zahl:

$$\underline{\boldsymbol{x}}^T \boldsymbol{b} = \underline{\boldsymbol{x}}^T \sum_{i=1}^{P} K_i^T \boldsymbol{b}_i = \sum_{i=1}^{P} (K_i \underline{\boldsymbol{x}})^T \boldsymbol{b}_i = \sum_{i=1}^{P} \underline{\boldsymbol{x}}_i^T \boldsymbol{b}_i. \tag{5.29}$$

Jede andere Vektorkombination benötigt eine Vektor-Typumwandlung evtl. mit Kommunikation. Das Produkt einer verteilten Matrix mit einem akkumulierten Vektor liefert einen verteilten Vektor. Aus der Definition 5.1 ergibt sich:

$$A\,\underline{\boldsymbol{x}} = \sum_{i=1}^{P} K_i^T A_i K_i \underline{\boldsymbol{x}} = \sum_{i=1}^{P} K_i^T \underbrace{A_i\,\underline{\boldsymbol{x}}_i}_{\boldsymbol{r}_i} = \boldsymbol{r}. \tag{5.30}$$

Die Kommunikation wird nur durch die Summe initiiert, wobei das lokale Produkt $A_i\,\underline{\boldsymbol{x}}_i = \boldsymbol{r}_i$ ohne Kommunikation durchgeführt wird. Das Produkt einer verteilten Matrix mit einem verteilten Vektor benötigt eine Vektorumwandlung, bevor die Multiplikation (5.30) ausgeführt werden kann.

Überlappende Partitionierung

Das Konzept von verteilten und akkumulierten Vektoren funktioniert auch im Fall der überlappenden Partitionen, obwohl die Teilmatrizen A_i neu definiert werden müssen, damit die Beziehung (5.26) stimmt. Mehrere Randelemente gehören in diesem Fall gleichzeitig zu mehr als einer Partition und werden parallel in die entsprechenden Teilgebietsmatrizen $\hat{A}_i$ assembliert. Um das Problem der mehrfachen Elemente-Assemblierung zu lösen, wird die Wertigkeit eines Elements W_k eingeführt, die gleich der Anzahl der das Element enthaltenden Teilgebiete ist. Dann gilt

$$A_i = \sum_{T_k \in \mathcal{T}_i} \frac{1}{W_k} L_k, \tag{5.31}$$

wobei T_k ein Element des Teilgitters $\mathcal{T}_i$ und L_k die entsprechende lokale Matrix sind.

Parallelisierung iterativer Verfahren

Als ein Beispiel betrachten wir eine Parallelisierung des klassischen CG-Verfahrens (Tab. 5.5). Die Parallelisierungsstrategie besteht in der Einteilung sämtlicher Vektoren in zwei o.g. Klassen: akkumulierte und verteilte Vektoren (Def. 5.1,5.2), wobei die Anzahl der notwendigen Kommunikationen möglichst klein gehalten werden soll.

	sequenziell	parallel	
1	Wähle $\boldsymbol{x}_0$	Wähle $\underline{\boldsymbol{x}}_0$	
2	$\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}_0$	$\boldsymbol{r} = \boldsymbol{b} - A\underline{\boldsymbol{x}}_0$	
3	$\boldsymbol{s}_0 = \boldsymbol{r}_0$	$\underline{\boldsymbol{s}}_0 = \boldsymbol{r}_0$	Komm (5.27)
4	$\delta_0 = \boldsymbol{r}_0^T \boldsymbol{r}_0$	$\delta_0 = \boldsymbol{r}_0^T \boldsymbol{r}_0$	
5	do $k = 0, 1, ...$	do $k = 0, 1, ...$	
6	$\boldsymbol{q}_k = A\boldsymbol{s}_k$	$\boldsymbol{q}_k = A\underline{\boldsymbol{s}}_k$	
7	$\alpha_k = \delta_k / \boldsymbol{q}_k^T \boldsymbol{s}_k$	$\alpha_k = \delta_k / \boldsymbol{q}_k^T \underline{\boldsymbol{s}}_k$	Komm (5.29)
8	$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{s}_k$	$\underline{\boldsymbol{x}}_{k+1} = \underline{\boldsymbol{x}}_k + \alpha_k \underline{\boldsymbol{s}}_k$	
9	$\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k \boldsymbol{q}_k$	$\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k \boldsymbol{q}_k$	
10		$\underline{\boldsymbol{r}}_{k+1} = \boldsymbol{r}_{k+1}$	Komm (5.27)
11	$\delta_{k+1} = \boldsymbol{r}_{k+1}^T \boldsymbol{r}_{k+1}$	$\delta_{k+1} = \underline{\boldsymbol{r}}_{k+1}^T \boldsymbol{r}_{k+1}$	Komm (5.29)
12	$\beta_k = \delta_{k+1} / \delta_k$	$\beta_k = \delta_{k+1} / \delta_k$	
13	$\boldsymbol{s}_{k+1} = \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{s}_k$	$\underline{\boldsymbol{s}}_{k+1} = \underline{\boldsymbol{r}}_{k+1} + \beta_k \underline{\boldsymbol{s}}_k$	
14	until $(\sqrt{\delta_{k+1}/\delta_0} < TOL)$	until $(\sqrt{\delta_{k+1}/\delta_0} < TOL)$	

Tabelle 5.5: Eine sequenzielle und eine parallele Version des klassischen CG-Verfahrens

Die Matrix A wird verteilt assembliert und gespeichert (5.26). Vektoren $\boldsymbol{s} \rightarrow \underline{\boldsymbol{s}}$, $\boldsymbol{x} \rightarrow \underline{\boldsymbol{x}}$ und $\underline{\boldsymbol{r}}$ sind akkumuliert. Vektoren $\boldsymbol{b}$, $\boldsymbol{q}$ und $\boldsymbol{r}$ sind verteilt gespeichert und werden im Iterationsverlauf nicht akkumuliert. Mit dieser Auswahl kann das Matrix-Vektor-Produkt ohne Kommunikation ausgeführt werden. Da in den beiden Skalarprodukten der CG-Iteration nun unterschiedliche Vektor-Typen auftreten, können diese Operationen mit einem geringen Kommunikationsaufwand durchgeführt werden. Der resultierende parallele CG-Algorithmus (Tab. 5.5, 2. Spalte) erfordert pro Iteration zwei ALL_REDUCE-Operationen [10, 74, 115, 116] mit einer reellen Zahl und eine Vektorakkumulation.

Die weiteren Iterationsverfahren (Tab. 5.4) werden genau so parallelisiert, indem zuerst alle Vektoren in die zwei Klassen so eingeteilt werden, dass möglichst wenig Kommunikation notwendig ist. Dabei entstehen die o.g. Kommunikations-Operationen: Vektorumwandlung und Skalarprodukt.

5.2.8 A posteriori Fehlerschätzung

In der praktischen Anwendung aller Diskretisierungsverfahren besteht eine wichtige Fragestellung darin, wie stark tatsächlich die berechnete Näherungslösung $(\boldsymbol{u}_h, p_h)$ von der zu approximierenden Lösung $(\boldsymbol{u}, p)$ abweicht. Dieses Konzept ist verantwortlich für die Messung des Fehlers anhand gewisser Normen. Der Fehler kann

auch bezüglich eines linearen Funktionals $J(\boldsymbol{u})$ abgeschätzt werden, das nach der physikalischen Bedeutung der Aufgabe von Interesse ist. In den Strömungsproblemen kann das z.B. die Flussfunktion sein. Wünschenswert sind also Fehlerabschätzungen der Form

$$D_1\eta \leq \|\boldsymbol{u} - \boldsymbol{u}_h\|_1 + \|p - p_h\|_0 \leq D_2\eta \tag{5.32}$$

oder

$$D_1\eta \leq |J(\boldsymbol{u}) - J(\boldsymbol{u}_h)| \leq D_2\eta \tag{5.33}$$

mit von Diskretisierungsparametern unabhängigen Konstanten $D_1, D_2 > 0$ und

$$\eta = \left(\sum_{K \in \mathcal{T}_h} \eta_K^2 \right)^{\frac{1}{2}}, \tag{5.34}$$

wobei die Größen η_K mit nicht zu großem Aufwand aus den auf dem Element K bekannten Daten berechenbar sein sollen. Die beidseitige Abschätzung in (5.32), (5.33) sind dazu nötig, eine Fehler-Überschätzung und dadurch evtl. zu starke Gitterverfeinerung und zusätzlichen Aufwand zu vermeiden. Der Quotient $\frac{D_2}{D_1}$ ist dann ein Maß für die ***Effizienz*** des Fehlerschätzers.

Die a posteriori Fehlerschätzer lassen sich in die folgenden Klassen einteilen:

1. Residuumsabschätzung: Schätze den Fehler der numerischen Lösung durch eine geeignete Norm ihres Residuums bezüglich der starken Form der Differenzialgleichung [174, 176, 36];
2. Lösung von lokalen Problemen: Löse lokale diskrete Probleme ähnlich, aber einfacher als das ursprüngliche Problem und verwende geeignete Norme zur Fehlerschätzung [9];
3. Hierarchisch basierte Fehlerschätzung: Berechne das Residuum der FE-Lösung bezüglich eines anderen FE-Raums bestehend aus den Elementen höherer Ordnung oder auf einem verfeinerten Gitter [11];
4. Mittelwert-Methode: Verwende eine Extrapolation oder Mittelwertbildung zur Fehlerschätzung [28, 29];
5. Dualgewichtete Residuumabschätzungen (DWR-Methode): Schätzt den Fehler in der Form (5.33) anhand primaler und dualer numerischer Lösungen der schwachen Form der Differenzialgleichung [15, 16, 17, 81, 140, 141].

Einen Vergleich und eine Diskussion der Fehlerschätzer (1-4) findet man auch bei Verfürth [175].

Wir begrenzen die Betrachtung auf ein konkretes Beispiel der Fehlerschätzung für die Navier-Stokes-Gleichungen, ohne zugehörige Herleitung, die in der jeweiligen Quelle angegeben ist. Die traditionelle Vorgehensweise zur residuumbasierten Fehlerschätzung führt zur folgenden Darstellung [177]:

$$\begin{aligned} \eta_K = \Bigg(& h_k^2 \| - \Delta \boldsymbol{u}_h + \nabla p_h + \frac{1}{\nu}(\boldsymbol{u}_h \cdot \nabla)\boldsymbol{u}_h - \frac{1}{\nu}\boldsymbol{f}\|^2_{L^2(K)} \\ & + \|\nabla \cdot \boldsymbol{u}_h\|^2_{L^2(K)} + \sum_{\substack{E \in \mathcal{E}_h \\ E \subset \partial K}} h_E \|[\boldsymbol{n}_E \cdot (\nabla \boldsymbol{u}_h - p_h I)]_E\|^2_{L^2(E)} \Bigg)^{\frac{1}{2}}. \end{aligned} \tag{5.35}$$

Hierbei bezeichnen I die $d \times d$ Einheitsmatrix, $\mathcal{E}_h$ die Menge der inneren Kanten in $\mathcal{T}_h$, $\boldsymbol{n}_E$ einen zur Kante E senkrechten Einheitsvektor und $[\phi]_E$ den Sprung von ϕ über E in Richtung $\boldsymbol{n}_E$. Die Kante E hat Länge h_E. Man beachte, dass die Terme $[\boldsymbol{n}_E \cdot p_h I]_E$ verschwinden, wenn wir eine stetige Druckapproximation verwenden.

Ähnlich zu den Konzepten **Gleichungssystemlöser** oder **Präkonditionierung** stellt jeder einzelne Fehlerschätzer eine getrennte Software-Komponente dar, die in der Kompilierungszeit statisch in einen Solver mit einbezogen werden kann.

Gitteradaption

Der lokale Fehler η_K kann durch Verfeinerung der Gitterelemente verbessert werden, die nach den folgenden Regeln abläuft:

- Gleichverteilungskriterium: Die Anpassung des Gitters (Verfeinerung oder Vergröberung von Elementen) erfolgt mit dem Ziel, dass die lokalen Fehlerschätzer η_K^* für das angestrebte neue Gitter $\mathcal{T}_h^*$ den gleichen Wert über alle Elemente K annehmen.

- Abschneidekriterium: Es werden nur jene Elemente K verfeinert, deren lokaler Schätzer η_K einen gewissen Mindestbeitrag zum Gesamtschätzer η liefert. Dieser Beitrag wird in der Form $\lambda\eta$ mit $\lambda \in (0, 1)$ als Parameter festgelegt.

- Reduktionskriterium: Es wird $\varepsilon_\eta = \lambda\eta$ gesetzt. Danach werden einige Anpassungsschritte nach dem Gleichverteilungskriterium mit der Toleranz ε_η durchgeführt.

Algorithmisch kann der Adaptionsprozess für stationäre Probleme so aussehen:

1. Erzeuge ein grobes Anfangsgitter $\mathcal{T}_0$, das das Problemgebiet Ω ausreichend gut darstellt, und setze $i = 0$;

2. Löse das diskrete Problem auf dem Gitter $\mathcal{T}_i$;

3. Berechne für jedes Element $K \in \mathcal{T}_i$ die a posteriori Fehlerschätzung η_K und $\eta_i = \max_{K \in \mathcal{T}_i} \eta_K$;

4. Falls der Fehler $\eta_i \leq TOL$ ist, STOP. Sonst verfeinere alle Elemente $K \in \mathcal{T}_i$ mit $\eta_K \geq \lambda\eta_i$. Verfeinere evtl. weitere Elemente, um ein zulässiges Gitter $\mathcal{T}_{i+1}$ zu erhalten. Setze $i := i + 1$ und gehe zum Schritt 2 zurück.

Für instationäre Probleme muss die Genauigkeit der berechneten numerischen Lösung jeweils nach einigen Zeitschritten abgeschätzt werden, wobei die räumliche Verfeinerung mit der Zeitschrittsteuerung gekoppelt sein soll. Eine partielle Vergröberung oder vollständig neue Gittergenerierung kann notwendig sein. In den beiden Fällen kann die Gitteradaption mit der Bewegten-Gitter-Technik gewechselt oder zusammen verwendet werden. Dabei werden einige Gitterpunkten verschoben, wobei deren Gesamtzahl konstant bleibt.

Die Gitteradaptions-Algorithmus als Software-Komponente kann in der Form einer Policy-Klasse (Alexandrescu [3], Kap. 1) implementiert und in eine FEM zusammen mit einem geeigneten Fehlerschätzer statisch eingebaut werden.

5.3 Implementierung von Komponenten und Datenstrukturen

Im Abschnitt 5.3.1 werden zuerst die Parameter des Domänenmodells mittels DSL-Sprache zusammengefasst. Die weiteren Abschnitte beschreiben eine Implementierung der ausgewählten Komponenten, die besonders wichtig für den gesamten Solver sind.

5.3.1 Vorläufige DSL

Die DSL ist eine domänenspezifische Sprache zur Beschreibung eines Simulationssolvers im Problemraum. Damit wird die aus Merkmaldiagrammen entstandene Konfiguration der Eigenschaften des Domänenmodells beschrieben (Tab. 5.6), obwohl keine Kenntnisse der Programmierdetails erforderlich sind. Die Beschreibung des gewünschten Solvers erfolgt auf verschiedenen Abstraktionsniveaus ausschließlich mit den im Problemraum gegebenen Anforderungen.

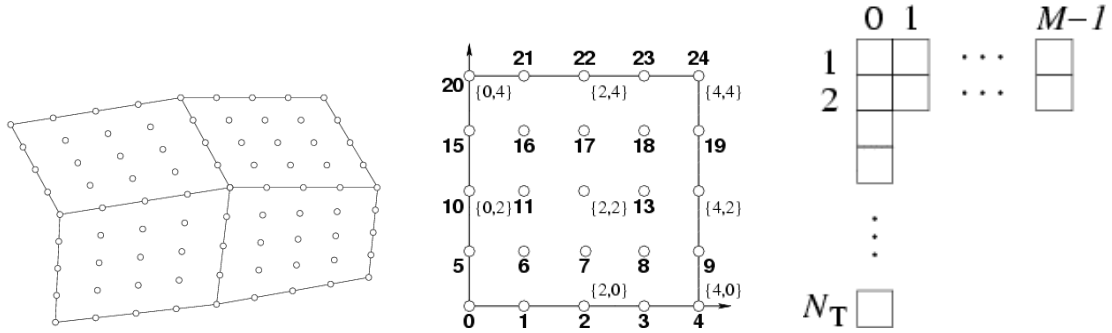
Solver	Solver[Import,Method,Result,Parallel]
Import	GAMBIT NEU-Format | ...
Result	DX-Format | ...
Parallel	true | false
Method	FEM[Equation,Mesh,StokesElement,LS-Solver,Assembling] | FVM[Equation,Mesh,Scheme,LS-Solver] | FDM[Equation,Mesh,Scheme,LS-Solver]
Equation	NavierStokes[TimeDiskr,Linearization] | Stokes[TimeDiskr] | SteadyStokes
LS-Solver	GMRES | TFQMR | BiCGStab | ...
TimeDiskr	ThetaScheme[Numerator,Denominator] | AdamsBashforth
Linearization	SemiExplicit | NewtonMethod
Mesh	F1[Dimension] | R1[Dimension] | R2[Dimension]
StokesElement	StableStokesElement[Element,Element] | UnstableStokesElement[Element,Element,Stabilization]
Element	Conform[Geometry,Dimension,Order] | CrouzeixRaviart[Geometry,Dimension] | RannacherTurek[Geometry,Dimension] | ...
Geometry	triangular | quadrilateral
Dimension	1 | 2 | 3

Tabelle 5.6: DSL-Grammatik

Die vorläufige DSL-Grammatik (Tab. 5.6) kann alle vorhandene Verfahren aus dem Problemraum nicht enthalten. Die Mengen von notwendigen Parameter sind an einigen Stellen unendlich, z.B. die Eingangs- (Import) und Ausgangs-Datenformate (Result). Es gibt viele andere nichtkonforme Stokes-Elemente, Stabilisierungstechniken, die hier nicht angegeben wurden. Die FVM- und FDM-Schemas wurden auch nicht als Parameter beschrieben. Die aufgebaute Architektur mit der Berücksichtigung von vielen Klassen von Verfahren gestattet aber das Domänenmodell durch die Einführung von neuen Parameter ohne Änderungen zu ergänzen.

5.3.2 Freiheitsgradnummerierung

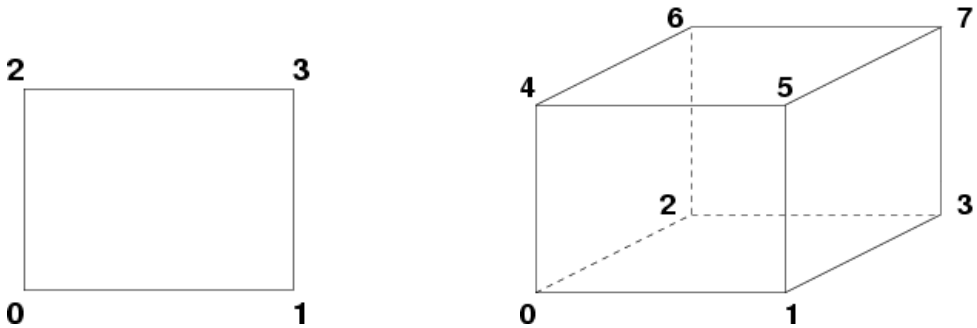
Wir befassen uns in diesem Abschnitt mit der Nummerierung von Freiheitsgraden ausgehend von einem Gitter im R1-Format (Abb. 5.5), obwohl die Implementierung auf andere Gitter-Datenstrukturen einfach übertragen werden kann. Wir beschränken uns auf den Fall konformer Ansatzfunktionen bzw. die entsprechende Freiheitsgrad-Verteilung auf einem Gitter-Element. Der Assemblierungsprozess läuft über Gitterzellen ab (Abschnitt 5.2.4). Das Ziel der Nummerierung ist aus der Gitter-Datenstruktur eine Tabelle zu bekommen, wobei jede Zeile $(1...N_T)$ einer Zelle entspricht und die globalen Freiheitsgradnummern speichert (Abb. 5.13,c). Jede Zeile trägt also die Indizes für die lokalen Steifigkeitsmatrizen.



a) b) c)

Abbildung 5.13: Freiheitsgradnummerierung auf einem Viereckgitter in 2D

Wenn man eine unvollständige Gitter-Datenstruktur nutzen möchte (Abb. 5.5, R1, R2), muss man zuerst ein Muster definieren, in welcher Reihenfolge die Ecken ein Gitter-Element bestimmen. Das ist insbesondere wichtig für ein Viereck-Gitter, wobei nicht jeweils zwei Ecken eine Kante bilden (Abb. 5.14). Nach einem solchen Muster können die Kanten im 2D-Fall (Kanten und Flächen im 3D-Fall) aus einer Zelle-Ecken-Liste problemlos wiederherstellt werden. Diese Muster-Nummerierung entspricht auch der lokalen Freiheitsgradnummerierung erster Ordnung. Da wir den Nummerierungsprozess für eine beliebige Ordnung m automatisieren wollen, werden die Nummern auf dieselbe Weise verteilt: Von links nach rechts und dann nach oben (Abb. 5.13,b); Im 3D-Fall: geht man lexikografisch entsprechend der Achsenbezeichnung vor.



a) Viereck b) Hexaeder

Abbildung 5.14: Ecken-Nummerierung in einem Viereck-Gitter

Zum Aufbau von Ansatzfunktionen zu jedem Freiheitsgrad werden die mehrdimensionalen Nummern benutzt (Abschnitt 4.3), deshalb muss die Abbildung von solchen Nummern auf die nichtnegativen ganzen Zahlen und umgekehrt $\{k_i, k_j\} \leftrightarrow \hat{N}$ gebildet werden. Die Abbildungsfunktion stammt aus den Formeln (4.10),(4.11) für die Dimension der lokalen Matrizen M bzw. die maximale lokale Nummer im Gitter-Element. In einem Viereckgitter gilt:

$$\hat{N}(d,m) = \sum_{i=1}^{d} k_i M(d-i,m) = \sum_{i=1}^{d} k_i (m+1)^{d-i}, \tag{5.36}$$

in einem Dreieckgitter dagegen

$$\hat{N}(d,m) = \sum_{i=1}^{d} \sum_{j=1}^{k_i} M(d-i, m-j+1). \tag{5.37}$$

Die Umkehr-Abbildung ist komplizierter. Für ein Viereckgitter muss $\hat{N}$ Modulo $(m+1)^{d-i}$ zur Bestimmung von k_i zerlegt werden. Für ein Dreieckgitter, aus $\hat{N}$ werden die Zahlen $M(d-i, m-j+1)$ subtrahiert, solange $\hat{N} \geq 0$ gilt, wobei $i = 1, ..., d$ und $j = 0, ..., m$ bezeichnet. Dann bestimmt die Anzahl der erfolgreichen j-Iterationen den Wert k_i. Damit ist die Frage der lokalen Freiheitsgradnummerierung abgeschlossen.

Wir wollen nun jedem Freiheitsgrad des gesamten Gitters eine eindeutige Nummer geben, wobei ein Freiheitsgrad sich in einer Ecke, auf einer Kante, auf einer Fläche oder im Zellen-Inneren befinden kann. Wenn eine Nummer einem Freiheitsgrad z.B. in der Gitter-Ecke zugeordnet und in die Tabelle eingetragen ist, muss die Nummer auch für alle benachbarten Zellen in die Tabelle in die richtige Spalte eingetragen werden. Die Spaltennummer bzw. die lokale Freiheitsgradnummer (z.B. in der Abb. 5.13,b) wird aus der lokalen Ecken-Nummer (Abb. 5.14) bestimmt. So entsteht eine gitterunabhängige Abbildung der Ecken-Nummern in die Freiheitsgradnummern $V_i \rightarrow F_i$ lokal auf einem Referenz-Element. Im Beispiel der Abbildung (5.13,b) mit der Ordnung $m = 4$ ist die Abbildung: $\{0,1,2,3\} \rightarrow \{0,4,20,24\}$. Ähnliche Abbildungen müssen noch zur richtigen Nummerierung von Kanten- und Flächen-Punkten aufgebaut werden. Die Schwierigkeit besteht aber darin, dass die Abbildung nicht nur die lokale Entity-Nummer, sondern auch die Entity-Orientierung berücksichtigen muss. Um die Entity-Orientierung zu unterscheiden werden Kanten und Flächen durch die Ecken bezeichnet: eine Kanten durch zwei Ecken, eine Fläche durch drei Ecken. Dann unterscheidet sich die Kante $(0,1)$ von der Kante $(1,0)$ und die Fläche $(0,1,2)$ von der Fläche $(0,2,1)$. Die Kanten $(0,1)$ und $(1,0)$ enthalten entsprechend innere Punkte $\{1,2,3\}$ und $\{3,2,1\}$ (Abb. 5.13,b), wobei die Reihenfolge wichtig ist. Für eine Fläche muss die Reihenfolge noch definiert werden. Wir nehmen an, dass die Nummerierung von der ersten zur zweiten Flächen-Ecke zeilenweise erfolgt. Dann enthalten die Flächen $(0,1,2)$ und $(0,2,1)$ entsprechend die Punkte $\{6,7,8,11,12,13,16,17,18\}$ und $\{6,11,16,7,12,17,8,13,18\}$. Mit diesen Regeln werden die Kanten- und Flächen-Abbildungen aufgebaut.

Die Gitter-Datenstruktur R1 (Abb. 5.5) stellt uns eine Zellen-Liste mit den für jede Zelle nach dem Muster (Abb. 5.14) geordneten Ecken und eine Ecken-Liste, die für jede Ecke die benachbarten Zellen enthält, zur Verfügung. Da wir von einer

unvollständigen Gitter-Datenstruktur ausgehen und eine zellenbezogene Tabelle als Resultat (Abb. 5.13,c) aufbauen wollen, lassen wir den Nummerierungsprozess nach Zellen hierarchisch ablaufen: Ecken, kanteninnere Punkte, flächeninnere Punkte und anschließend zelleninnere Punkte.

- Der Zähler bekommt den Anfangswert.
- Nummerierung von Ecken:
  1. Die erste Zelle wird die aktuelle.
  2. Jede noch nicht nummerierte Ecke der aktuellen Zelle bekommt die nächste Nummer. Diese Nummer wird den Nachbarnzellen der Ecke übergeben und dort mit der Abbildung der richtigen Ecke zugewiesen, wobei die Nachbarnzellen nur mit einer größeren Nummer als die aktuelle Zelle berücksichtigt werden müssen, weil die Ecken der vorhergehenden Zellen schon vollständig durchnummeriert sind.
  3. Die nächste Zelle wird die aktuelle und Schritt 2 wird wiederholt.
- Nummerierung von Kanten:
  1. Die erste Zelle wird die aktuelle.
  2. Jeder noch nicht nummerierten Kante A_iA_j der aktuellen Zelle werden die nächsten $m-1$ Nummern zugeordnet. In den Nachbarnzellen der Ecke A_i werden die Kanten mit der Ecke A_j gesucht. Falls eine solche Kante gefunden ist, werden die neuen Nummern mittels der Abbildung für die Nachbarzelle eingetragen. Die Nachbarzellen müssen dabei auch nur mit einer größeren Nummer als die aktuelle Zelle betrachtet werden, weil die Kanten der vorhergehenden Zellen schon vollständig durchnummeriert sind.
  3. Die nächste Zelle wird die aktuelle und Schritt 2 wird wiederholt.
- Nummerierung von Flächen (nur in 3D-Fall):
  1. Die erste Zelle wird die aktuelle.
  2. Jeder noch nicht nummerierten Fläche $A_iA_jA_k$ der aktuellen Zelle werden die nächsten $M(d-1, m-2)$ im Viereck-Fall oder $M(d-1, m-3)$ im Dreieck-Fall Nummern zugeordnet. In den Nachbarnzellen der Ecke A_i werden die Flächen mit den Ecken A_j und A_k gesucht. Falls eine solche Fläche gefunden ist, werden die neuen Nummern mittels der Abbildung für die Nachbarnzelle eingetragen. Die Nachbarnzellen müssen dabei auch nur mit einer größeren Nummer als die aktuelle Zelle betrachtet werden.
  3. Die nächste Zelle wird die aktuelle und Schritt 2 wird wiederholt.
- Nummerierung von Punkten im Zellen-Inneren. Dabei werden alle Zellen durchlaufen und die neuen Nummern vergeben. Keine Nachbarsuche ist erforderlich.

Für das Stokes-/Navier-Stokes-Problem muss der Algorithmus doppelt ausgeführt werden, für die Geschwindigkeit und für den Druck. Die Nummerierung für die Geschwindigkeit erfolgt mit dem Schritt 1,2 oder 3 entsprechend der Anzahl der

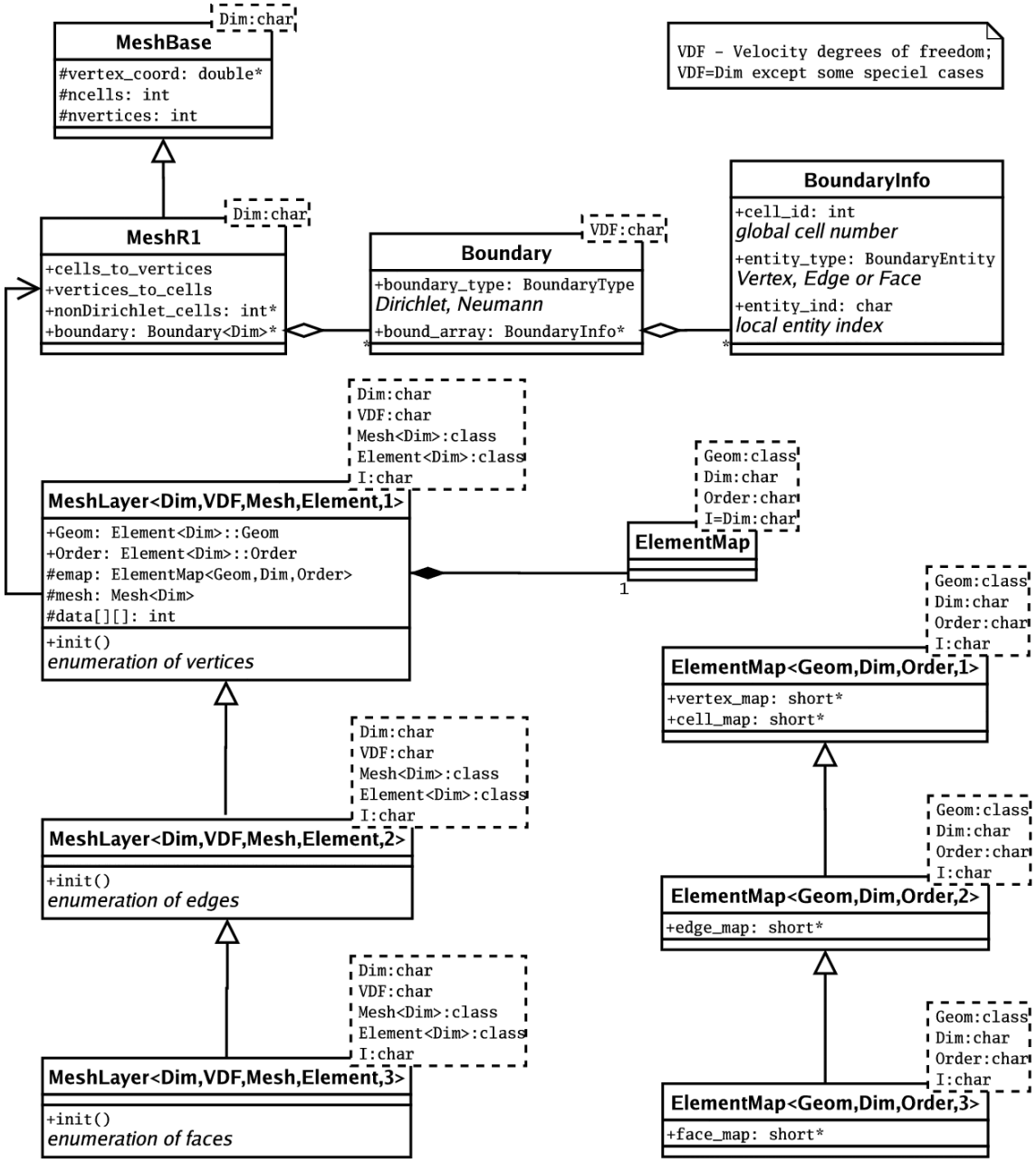
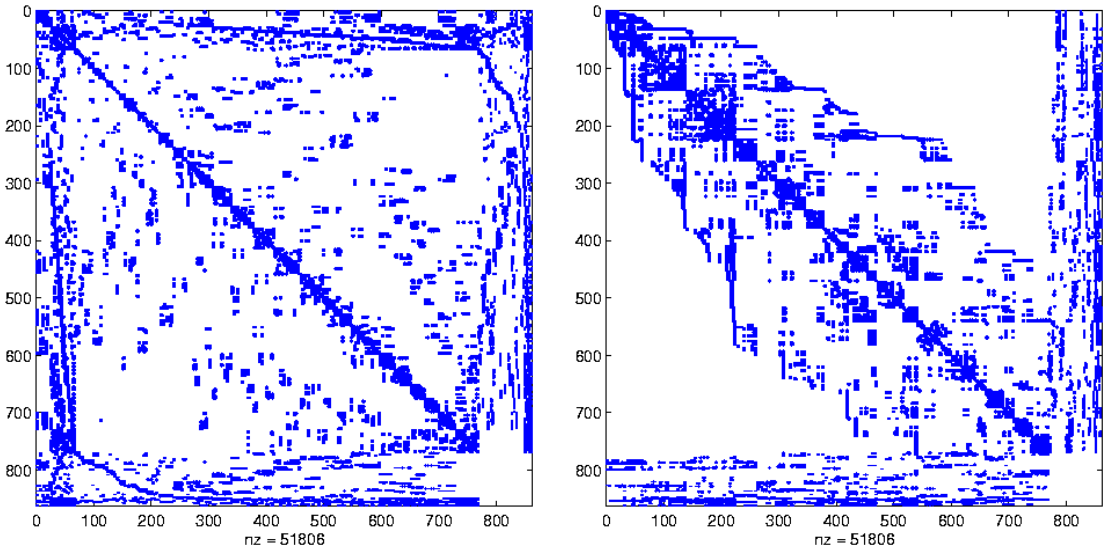


Abbildung 5.15: UML-Diagramm der Gitter-Datenstrukturen

Geschwindigkeitskomponenten. Der Zähler für den Druck bekommt logischerweise die gesamte Anzahl der Geschwindigkeitsfreiheitsgrade als Anfangswert.

Die Abbildung 5.15 stellt ein UML-Klassendiagramm zur Implementierung der Gitterdatenstruktur R1 und die Freiheitsgradnummerierung. Das Klassen-Template `MeshR1` hat nur die Dimension d als Parameter, enthält die Zellen-Ecken und Ecken-Zellen-Beziehungen und erbt das Klassen-Template `MeshBase`, das die gemeinsamen Daten für alle Gitter-Datenstrukturen trägt: Ecken-Koordinaten, Ecken- und Zellen-Anzahl. Das Klassen-Template `MeshR1` enthält auch ein Array der Randgruppen `Boundary`. Eine Randgruppe sammelt die Randecken von gleichem Typ, wie z.B. Dirichlet- oder Neumann-Randecken. Jede Randecke ist in der Beziehung (5.1) in

der Klasse `BoundaryInfo` gespeichert. Die lokale Abbildungen werden im Klassen-Template `ElementMap` implementiert. Dieses Template hat keine allgemeine Definition ist aber für die Dimensionen 1,2 und 3 spezialisiert, wobei jede Spezialisierung die Spezialisierung für die vorhergehende Dimension erbt, d.h., die erste Spezialisierung enthält nur die Ecken- und Zellen-Abbildung, die zweite - die Kanten-Abbildung und die dritte - die Flächen-Abbildung. Damit werden nur die notwendigen Abbildungen für die bestimmte Dimension zusammengestellt. Die Nummerierung erfolgt im Klassen-Template `MeshLayer`, das eine ähnliche hierarchische Gestalt bezüglich der Dimension hat. Alle Daten befinden sich aber in der ersten Spezialisierung, wobei die Nummerierungsalgorithmen über die Hierarchie verteilt sind: Die erste Spezialisierung - die Nummerierung von Ecken, die zweite - Kanten, die dritte - Flächen. Eine kurze Notationsbeschreibung von UML-Klassendiagrammen gibt es im Anhang A.2.2.



a) ohne Bandbreitereduktion b) mit Cuthill-McKee-Reduktion

Abbildung 5.16: Struktur der schwachbesetzten Steifigkeitsmatrix ohne und mit der Bandbreite-Reduktion mittels Cuthill-McKee-Algorithmus, entstanden durch die P_2/P_1-GFEM-Diskretisierung der instationären Navier-Stokes-Gleichungen auf einem Tetraeder-Gitter.

Die entstehende Freiheitsgradnummerierung führt zu einer großen Bandbreite der Steifigkeitsmatrix (Abb. 5.16a), da der Nummerabstand zwischen Ecken, Kanten und Flächen groß ist. Für die Bandbreitereduktion kann der Cuthill-McKee-Algorithmus angewendet werden (Abb. 5.16). Die dagegen unvermeidbare Umnummerierung der Freiheitsgrade muss für die Dirichlet-Randpunkten durchgeführt werden, um schließlich das lineare Gleichungssystem in der Form (5.15) zu erhalten, wobei zuerst die Geschwindigkeit $u_1, v_1, w_1, u_2, v_2, w_2, ...$, dann der Druck $p_1, p_2, ...$ und am Ende die Dirichlet-Randpunkte $u_{D1}, v_{D1}, w_{D1}, u_{D2}, v_{D2}, w_{D2}, ...$ nummeriert werden. Alle Zellen müssen auch dementsprechend für die Assemblierung in die zwei Klassen, ohne und mit Dirichlet-Punkten eingeteilt werden.

5.3.3 Lineare-Algebra-Bibliotheken

Um die Assemblierung mit den lokalen Matrizen (Abschnitt 5.2.3) sowie die iterativen Verfahren (Abschnitt 5.2.5) einfach und effizient implementieren zu können, braucht man eine geeignete Lineare-Algebra-Bibliothek. In der Programmiersprache C++ muss eine solche Bibliothek sowohl über die flexiblen Matrix-Vektor-Ausdrücke im oberen Abstraktionsniveau als auch über eine leistungseffiziente Implementierung der Grundoperationen verfügen. Außerdem sind folgende unterschiedlichen Matrix-Formate notwendig:

- Vollbesetzte allgemeine ganzzahlige und Gleitkomma-Matrizen (lokale Matrizen);
- Vollbesetzte symmetrische ganzzahlige und Gleitkomma-Matrizen (lokale Matrizen);
- Schwachbesetzte Matrizen z.B. im CSR-Format (globale Matrix);
- Ein spezielles indiziertes Format: Eine vollbesetzte allgemeine oder symmetrische Matrix mit Zeilen- und Spalten-Indizes (lokale Matrizen). Dieses Format gehört nicht zu den in vielen Bibliotheken vorhandenen Standardformaten.

Die Matrix-Bibliothek GMCL ([125], [38],Kap.10) stellt mit Hilfe der generativen Programmiertechniken die größte Anzahl der Matrixformate zur Verfügung. Insgesamt 1840 verschiedene Matrixformate werden mittels der statischen Konfiguration generiert. Es gibt aber keine prozessor- und cache-orientierten Optimierungen in dieser Matrix-Bibliothek. Das spezielle indizierte Matrix-Format muss noch hinzugefügt werden.

Die LASC-Bibliothek [118, 121] wurde insbesondere für anspruchsvolles wissenschaftliches Rechnen im Rahmen eines Strömungssimulation-Solvers entwickelt. Auf dem oberen Niveau sind die Matrix-Vektor-Operationen durch die Expression-Templates-Technik optimiert. Für die Grundoperationen können sowohl eine effiziente C++-Implementierung als auch hardwareabhängige höchstperformante BLAS-Routinen [20, 131] eingesetzt werden. Solche Routinen werden häufig von Hardware-Lieferanten speziell entwickelt. Die Matrixformate können in dieser Bibliothek nicht konfiguriert und müssen einzeln implementiert werden. Deshalb wäre ein Einsatz der GMCL-Matrixgeneratoren in der LASC-Bibliothek von Vorteil.

Eine notwendige Erweiterung zu einer Matrix-Bibliothek besteht aus dem Satz von iterativen Gleichungssystemlöser (Abschnitt 5.2.5) und Präkonditionierer (Abschnitt 5.2.6), die nichts von der inneren Matrix-Darstellung wissen und ein beliebiges Matrix-Format einheitlich nutzen können.

Loop-Unrolling

Auf dem unterem Implementierungs-Niveau kann die Template-Metaprogrammierung (Abschnitt 4.1) auch für ein manuelles Loop-Unrolling erfolgreich eingesetzt werden. Solche Verfahren wurden gewöhnlich auf kleine Vektoren/Matrizen angewendet, deren Dimension eine statische Konstante ist und als ein Template-Parameter übergeben werden kann. Verschiedene Operationen mit solchen kleinen Vektoren können dann durch eine Template-Rekursion in sequenzielle Ausdrücke transformiert werden, die häufig schneller als Schleifen ausgeführt werden. Betrachten wir

beispielsweise die Summe von zwei Vektoren $\boldsymbol{x} = (x_1, x_2, x_3)^T$ und $\boldsymbol{y} = (y_1, y_2, y_3)^T$, so wird statt der Schleife

```
for (int i=0; i<3; ++i) res[i] = x[i] + y[i];
```

nach der Kompilierung sequenziell berechnet

```
res[0] = x[0] + y[0];
res[1] = x[1] + y[1];
res[2] = x[2] + y[2];
```

Diesen Rechen-Beschleunigungseffekt kann man dadurch erklären, dass viele moderne Prozessoren mehr als eine Rechen-Einheit beinhalten, die aber vom Programmcode nicht immer vollständig benutzt werden können. Der Pentium4-Prozessor hat z.B. zwei Rechen-Einheiten, der IBM-Power4-Prozessor - vier Rechen-Einheiten. Einige spezielle Rechen-Prozessoren, wie z.B. von Hitachi oder NEC, können viel mehr Rechen-Einheiten enthalten.

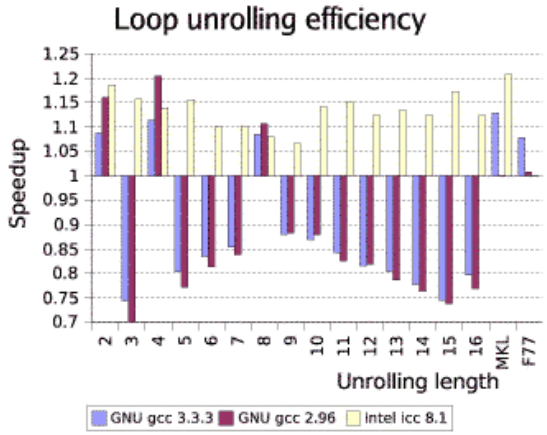
Wir interessieren uns insbesondere für die Skalarprodukt-Operation, die der Bestandteil des Matrix-Vektor-Produkts ist und bei der iterativen Gleichungssystemlösung am häufigsten vorkommt. Das Skalarprodukt von Vektoren $\boldsymbol{x}$ und $\boldsymbol{y}$ wird in einen einzigen Ausdruck mit Hilfe des Template-Metaprogramms (Anhang A.4.2, Listing A.11) während der Kompilierung transformiert:

```
res = x[0]*y[0] + x[1]*y[1] + x[2]*y[2];
```

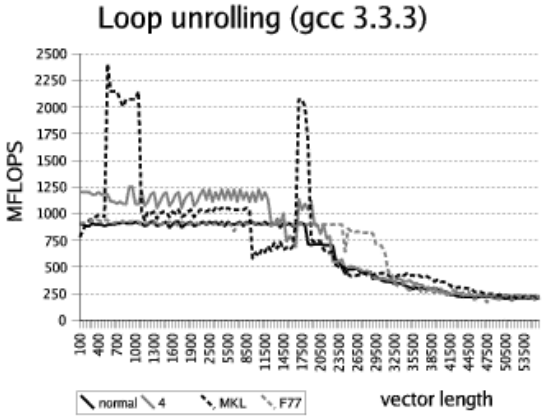
wobei die Vektorlänge N als Template-Parameter vor der Kompilierung angegeben werden muss. Diese Unrolling-Technik kann aber auch auf beliebige Vektoren mit einer nicht konstanten Länge angewendet werden, indem der Vektor auf kleine Stücke der Länge N zum Unrolling zugeschnitten wird. Erwartungsgemäß könnte diese Strategie auch CPU-Zeitgewinn bringen.

Die speziell entwickelte Benchmarktests der durch Unrolling-Strategie berechneten Skalar-Produkte von zwei zufällig generierten Vektoren liefern die Rechenleistung in den normalisierten MFLOPS (Abb. 5.17). Die Vektorlänge ändert sich von 10 bis 100000 Elementen und $N = 2, 3, ..., 16$. Die Tests wurden mit drei unterschiedlichen Compilern übersetzt, um die Compiler-Abhängigkeit des Effekts gleichzeitig zu untersuchen. Die Zusammenfassung der Tests wird in der Form von Beschleunigungs-Koeffizienten bezüglich des gewöhnlichen Schleifen-Codes dargestellt (Abb. 5.17,a). Zum Vergleich werden auch dieselben Tests mit der prozessoroptimierten MKL-Bibliothek [114] und mit der Fortran-Implementierung durchgeführt. Die Tests mit den größten Koeffizienten in der Zusammenfassung sind ausführlich in Abhängigkeit von der Vektorlänge angegeben (Abb. 5.17,b,c,d). Die höchsten MFLOPS-Gewinne gibt es für mittlere Vektorlängen (ungefähr bis 15000), wenn die beiden Vektoren vollständig in den Cachespeicher hineinpassen. Genau solche Vektoren werden im Kontext der lokalen Matrizen ($M \times M$) und der Zeilen der globalen schwachbesetzten Matrix behandelt. Für kleine Vektoren (10 bis 1000) wurde auch das Unrolling über die gesamte Länge zum Vergleich durchgeführt (Abb. 5.17,e,f), die aber keine guten Resultaten zeigte. Das stückweise durchgeführte Unrolling hat deutlich bessere MFLOPS-Werte.

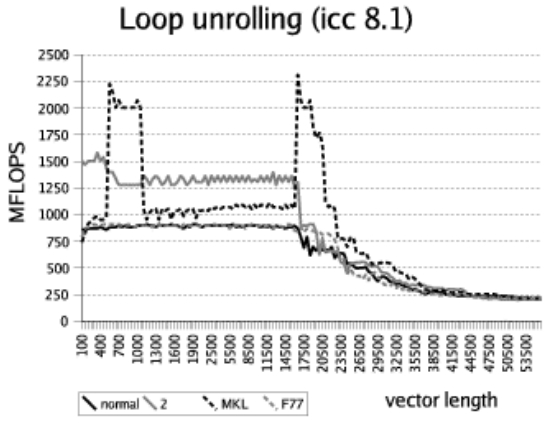
Eine Definition für die normalisierten MFLOPS sowie Benchmarktests für weitere Prozessoren findet man im Anhang A.3.



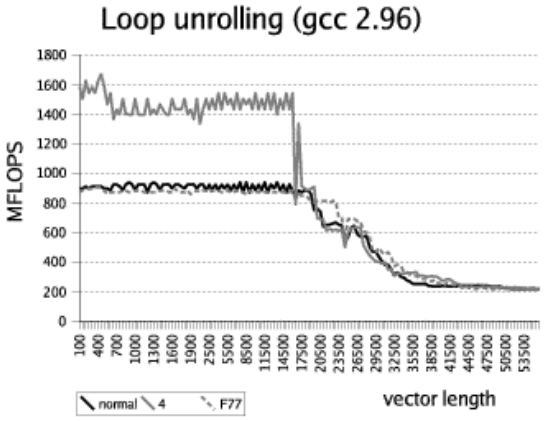
a) Beschleunigungs-Koeffizienten



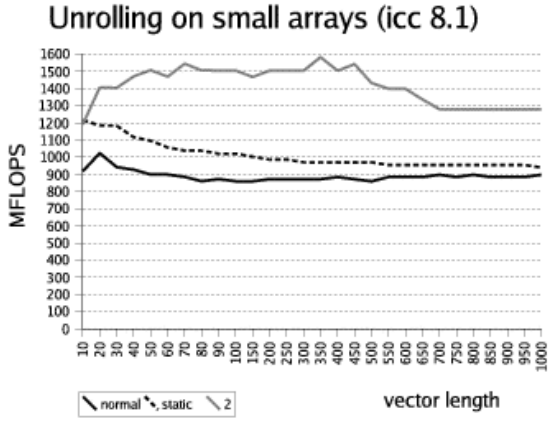
b) GNU gcc 3.3.3 Compiler



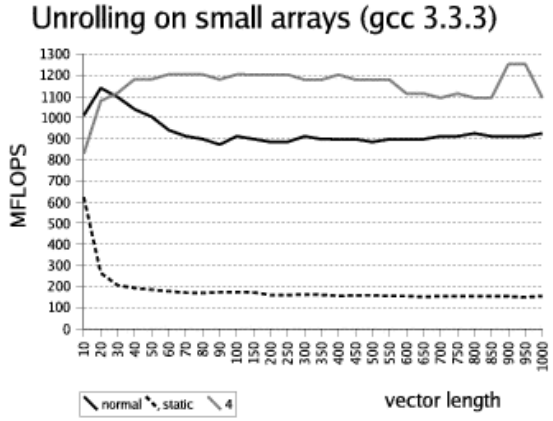
c) Intel icc 8.1 Compiler



d) GNU gcc 2.96 Compiler



e) Kleine Arrays (icc 8.1)



f) Kleine Arrays (gcc 3.3.3)

Abbildung 5.17: Loop-Unrolling durch Expression-Templates auf einem Pentium4-Prozessor

5.3.4 Objektfabriken

Wir gehen nun von den Optimierungsdetails auf dem unteren Entwicklungsniveau zum Aufbau eines Softwaresystem im Problembereich 'Strömungssimulations-Solver' aus den entwickelten Komponenten im Rahmen von Application-Engineering. Wir wollen keine Bibliothek bereitstellen, in der ein Nutzer die eine oder die andere Komponente oder deren Kombination auswählen kann, sondern ein Softwaresystem, das alle vorhandenen und korrekten Komponenten-Konfigurationen einschließt und dadurch die Verfahrensvielfalt zur Strömungssimulation zur Verfügung stellt. Das bedeutet, alle diese Konfigurationen werden ins Softwaresystem kompiliert, wobei die notwendigen Objekte erst in der Laufzeit nach der Benutzeranfrage erzeugt werden müssen. Das ist die Aufgabe einer sog. ***Objektfabrik***. Der Begriff stammt aus dem Buch von Gamma et al [62]. Eine Objektfabrik enthält Information über vorhandene Datentypen und erzeugt nach Anfrage ein Objekt eines Typs als ein konkretes Produkt.

Wir betrachten das folgende Beispiel: Sei eine FEM-Implementierung von der diskreten Gleichung, dem Gitter und dem Stokes-Element abhängig. Wir verwenden eine zur Sprache C++ ähnliche Notation:

```
FEM<Equation , Mesh , Element>
```

Jeder der drei Parameter kann variiert werden und stellt damit eine andere FEM dar. Wenn eine gleiche Operation mit allen solchen Objekten durchgeführt werden muss, kann es mit Hilfe von rekursiven Klassen-Templates, wie z.B. im Abschnitt 4.1.4, implementiert werden. Es ist aber unwahrscheinlich, dass ein Benutzer alle diese FEM-Versionen gleichzeitig benötigt, selbst wenn die Ressourcen dafür ausreichen würden.

Ein bestimmtes Objekt muss also nach Bedarf in der Laufzeit erzeugt werden. Diese Operation wird in der Sprache C++ normalerweise mit Hilfe des Operators `new` durchgeführt, wobei der Objekttyp schon vor der Kompilierung fest eingegeben werden muss; er kann in der Laufzeit als Parameter nicht eingesetzt werden. Das Problem liegt tiefer im Ursprung von Datentypen bzw. Klassen und von Objekten. Klassen und Objekte in der Sprache C++ sind zwei komplett verschiedene Bestandteile. Klassen werden vom Programmierer und Objekte von Programmen erstellt. Eine neue Klasse kann man in der Laufzeit nicht kreieren, genau wie man ein Objekt in der Kompilierungszeit nicht kreieren kann. Keine Klasse kann man kopieren, in einer Variable speichern oder mit einer Funktion zurückgeben. Es gibt Programmiersprachen, wo Klassen als Objekte angenommen werden und damit in der Laufzeit erstellt werden können. Solche dynamische Sprachen verlieren an die Typsicherheit und Performance bei mehr Flexibilität. In der Sprache C++ ist das statische Typsystem dagegen eine wichtige Grundlage der Codeoptimierung.

Die Entwicklung einer Objektfabrik ist in der Sprache C++ eine komplizierte Aufgabe, die durch Objekt-Polymorphismus bzw. virtuelle Funktionen realisierbar ist. Zuerst wird aber die Funktionalität einer Objektfabrik zusammengefasst:

- Die Klassen-Information wird in die Objektfabrik in der Form von Paaren $(id, creator)$ eingetragen. Die eindeutige Typ-Identifikationsnummer id muss in jeder Klasse vom Programmierer vorgesehen werden, weil kein Typidentifikator in der Laufzeit vorhanden ist. Wenn die Klassenmenge für die Ob-

jektfabrik unbekannt oder ganz flexibel ist, kann die Identifikationsnummer als Zufallszahl mit einer sehr kleinen Wiederholungswahrscheinlichkeit generiert werden. Der Zeiger *creator* speichert eine Funktion (z.B. die Funktion `Create()` im Listing 5.1), die auch Produkt-Hersteller heißt und ein Objekt einer bestimmten Klasse erzeugen kann.

- Ein Klassen-Eintrag kann nach der eindeutigen Identifikationsnummer *id* in der Objektfabrik gelöscht werden.
- Nach der Eingabe der Identifikationsnummer *id* kann ein Objekt einer bestimmten Klasse mit Hilfe der Funktionszeiger *creator* erzeugt werden, wenn das zugehörige Paar $(id, creator)$ in der Objektfabrik gefunden wurde.

Zur Unterstützung dieser Fabrik-Funktionalität benötigen alle für die Objektfabrik vorgesehenen Klassen (in unserem Beispiel FEM-Klassen) eine kleine Vorbereitung. Sie werden mit einer Grundklasse verallgemeinert. Eine solche abstrakte Klasse `AbstactMethod` (Listing 5.1) definiert alle für die FEM-Klassen gemeinsame Funktionen als leere virtuelle Funktionen. Das FEM-Klassen-Template erbt diese abstrakte Klasse und implementiert diese Funktionen. Die Objektfabrik erzeugt ein Objekt der FEM-Klasse mit Hilfe des gespeicherten Zeiger auf die Funktion `Create()` und gibt aber einen Zeiger auf das Objekt der abstrakten Klasse `AbstractMethod` zurück. Mit diesem Zeiger kann man nun die als virtuell deklarierte Funktionen aufrufen, wobei deren Implementierung aus der bestimmten FEM-Klasse durch Virtuellfunktionen-Mechanismus genommen wird. Genau so virtuell muss der Destruktor `~AbstractMethod` (Listing 5.1) definiert werden, um das Objekt der FEM-Klasse korrekt löschen zu können bzw. den gebrauchten Speicher frei zu geben. Auf dieselbe Weise können Objekte anderer FEM-Klassen auf Anfrage erzeugt und gelöscht werden. Eine vollständige Implementierung und Erklärung zu einer Objektfabrik findet man im Buch von Alexandrescu [3] und in der zugehörigen Bibliothek *Loki*.

Als virtuell müssen allerdings nur 'schwere' Funktionen definiert werden, deren Ablauf viel mehr CPU-Zeit benötigt als ein Funktions-Aufruf. Für jede virtuelle Funktion wird eine Aufruf-Tabelle in der Laufzeit generiert, die den Aufruf langsamer im Vergleich zu einer normalen Funktion macht ([171], Kap. 1.3). Die virtuellen Funktionen werden auch nie eingebaut (engl. inlined). In vielen Fällen können virtuelle Funktionen mit Templates-Techniken ersetzt werden (z.B. der Barton-Nackman-Trick [14]). Deshalb wurden virtuelle Funktionen im numerisch aufwendigen Code nirgendwo angewendet. In einer Objektfabrik helfen sie dagegen, den Objekt-Klasse-Abgrund zu überbrücken.

Zusammengefasst ist eine Objektfabrik der Hauptbestandteil eines Softwaresystems, das viele generative Komponenten einschließt, wobei deren Auswahl erst in der Laufzeit stattfinden muss. Der Ablauf eines solchen Softwaresystems besteht aus folgenden Schritten:

- Initialisierung der Objektfabrik.
- Registrierung aller notwendigen Klassen in der Objektfabrik durch die Paare $(id, creator)$. Das wird mit Hilfe der rekursiven Klassen-Templates implementiert, wobei alle zu registrierende Klassen zuerst in einer Typliste aufgelistet werden. Beispiele für solche Metaprogramme gibt es im Abschnitt 4.1.

- Erzeugung eines Objekts, d.h. eines Produkts der Objektfabrik; Nutzung dieses Objekts.
- Zerstörung des Objekts.

Die letzten zwei Schritte können mehrfach auch gleichzeitig wiederholt werden und sind unabhängig voneinander. Das liefert dem Nutzer eine hohe Flexibilität: Man kann je nach Aufgabengröße und Rechen-Ressourcen-Verfügbarkeit unterschiedliche und/oder mehrere FEM-Berechnungen gleichzeitig oder aufeinanderfolgend ausführen.

```
1  class AbstractMethod {
   public:
3      virtual void initialize() = 0;
       virtual void solve() = 0;
5      virtual void save_results() = 0;

7      virtual ~AbstractMethod() {}
   };
9
   template<class Equation, class Mesh, class Element>
11 class FEM : public AbstractMethod {
       static AbstractMethod* Create() {
13         return new FEM<Equation,Mesh,Element>;
       }
15     // weitere Implementierung
       //       . . .
17 };
```

Listing 5.1: Anpassung der FEM-Klasse zur Objektfabrik

Kapitel 6

Simulationen

What you see is all you get.
- Brian Kernigham

In diesem Kapitel werden die numerischen Experimente, die mit dem entwickelten Simulations-Solver durchgeführt wurden, kommentiert. Sie sind einerseits die Lösungen physikalischer Strömungsprobleme, andererseits die Untersuchung der verwendeten numerischen Verfahren auf Konvergenz-, Robustheit- und andere Eigenschaften. Für den letzten Aspekt der numerischen Experimente interessieren wir uns insbesondere, weil dadurch die Verfahren bzw. die entsprechenden Software Komponenten mit Hilfe bekannter Testprobleme ausgewählt, untersucht und angepasst werden können.

Gewöhnlicherweise wurde jede neuentwickelte Komponente des Solvers mit den einfachsten Strömungs-Beispielen in zwei und drei Dimensionen auf strukturierten und unstrukturierten Dreieck- und Viereckgittern so getestet, dass die Ergebnisse die realistischen Strömungsprozesse wiederspiegeln. Diese Entwicklungsphase wird hier ausgelassen.

Zuerst werden die Verfahren und Einstellungen des implementierten Simulations-Solvers abgesprochen. Als erstes Beispiel wird die Simulation der einfachsten zweidimensionalen Kanalströmung dargestellt, wobei auch eine analytische Lösung zum Vergleich angegeben werden kann (Abschnitt 6.2). Danach wird das Testbeispiel einer zweidimensionalen instationären Zylinderumströmung auf Dreieck- und Viereckgittern mit unterschiedlichen Zeitdiskretisierungsschemata untersucht (Abschnitt 6.3.1). Ähnlich ist die nachfolgende dreidimensionale Umströmung eines rechteckigen Hindernisses (Abschnitt 6.3.2). Verfahren höherer Ordnung werden auf den Quadrat- und Dreieck-Beispielen der Nischenströmung getestet (Abschnitt 6.4). Zum Schluß wird eine Simulation der Stokes-Strömung in einer komplexen Pipeline durchgeführt (Abschnitt 6.5).

6.1 Allgemeine Voraussetzungen

Die erste Version des entwickelten Strömungssimulations-Solver funktioniert folgendermaßen: Die inkompressiblen Navier-Stokes-Gleichungen und deren Sonderfälle (wie Stokes-Gleichungen, Abschnitt 6.5) werden mit einer Methode der Einschritt-θ-Version diskretisiert. Für den nichtlinearen Term wird entweder die Newton-Iteration

oder die semi-explizite Behandlung angewendet. Die Gleichungen werden weiter im Raum mittels GFEM auf allgemein unstrukturierten Dreieck- und Viereckgittern mit den konformen P_i/P_j- bzw. Q_i/Q_j-Stokes-Elementen diskretisiert, wobei nur die LBB-stabilen Kombinationen ($i > j$, $i \geq 2$, $j \geq 0$) gewählt werden.

Die globale Steifigskeitsmatrix wird nach den im Abschnitt (5.2.4) beschriebenen Strategien als Summe der lokalen Matrizen assembliert. Die lokalen Matrizen wurden einmal durch exakte Integration berechnet und abgespeichert.

Bei der Lösung des linearen Gleichungssystems mit der Matrix in Blockgestalt (5.17) wird der Präkonditionierer der Form (5.18) für die Stokes-Gleichungen und der Form (5.21) für die Navier-Stokes-Gleichungen mit X_M benutzt. Eine Anwendung der beiden Präkonditionierer erfordert eine Approximation der Matrizen F^{-1} und X_M^{-1} bzw. Lösung der linearen Gleichungssysteme

$$F\boldsymbol{x} = \boldsymbol{b}_1, \qquad X_M\boldsymbol{y} = \boldsymbol{b}_2$$

in jeder Iteration des präkonditionierten Verfahren. Das zweite System ist symmetrisch und wird mit dem CG-Verfahren gelöst. Im Fall der Stokes-Gleichung wird das erste System mit dem MINRES-Verfahren gelöst, weil die Matrix F symmetrisch und indefinit ist. Das MINRES-Verfahren läuft schneller und benötigt die gleiche Zeit für jede Iteration im Unterschied zu GMRES-Verfahren, das im Fall der Navier-Stokes-Gleichungen zur Approximation von F verwendet wird. Die CG-, MINRES- und GMRES-Verfahren haben ein monotones Konvergenzverhalten, deshalb werden sie auf 30 Iterationen begrenzt, wenn die Genauigkeit 10^{-3} für MINRES und GMRES oder 10^{-5} für CG schon vorher nicht erreicht wurde. Die globale Blockmatrix wird mit dem FGMRES-Verfahren [144] gelöst, das eine für eine Präkonditionierung günstige Modifikation des GMRES-Verfahrens darstellt.

Alle Simulationen zeitabhängiger Probleme beginnen mit einer Lösung des stationären Stokes-Problems auf dem gleichen Gitter und mit den gleichen Randbedingungen, damit die Anfangsbedingungen die Kontinuitätsgleichung erfüllen und dadurch die ganze Aufgabe korrekt gestellt ist (siehe auch [70], Abschnitt 3.9).

6.1.1 Pre- und Postprozessor

Abbildung 6.1: Pre- und Postprozessor

Alle in diesem Kapitel benutzte Gitter wurden mit einem kommerziellen Gittergenerator ***GAMBIT*** der Firma FLUENT [61] erstellt. Nach der Generierung wird das Gitter in einem textbasierten Austauschdatenformat (NEU-Format) abgespeichert, dann mit einem Hilfsprogramm in das Simulations-Solver-Format umgewandelt und für die Tests benutzt. Dadurch ist der Solver unabhängig von einem Gittergenerator. Der Einsatz eines anderen Gittergenerators erfordert also die Entwicklung eines Format-Umwandlungsprogramms.

Eine Darstellung von Resultaten wird mit dem freiverfügbaren ***Data Explorer*** (OpenDX) [126] durchgeführt, der sich schon in vielen anspruchsvollen Anwendungen bewährt hat. Dafür werden zuerst die Resultate sämtlicher Zeititerationen im DX-Format in Form eines binären Files `.dx.bin` und eines Text-Files `.dx` vom Solver gespeichert. Außerdem wurde ein Macro-Programm und ein Benutzer-Interface mit mehreren Funktionen innerhalb des Data Explorer entwickelt, die insbesondere die grafische Darstellung von instationären Strömungsvorgängen gut realisieren. Darunter sind die folgenden Darstellungs-Optionen:

- Gitter, Randgitter und Freiheitsgrade;
- Geschwindigkeits-Vektorfelder sowohl der vollständigen Vektoren als auch deren Projektionen auf eine Achse;
- Geschwindigkeits- und Druck-Konturen oder Farbfelder, die in 3D auf einem ausgewählten Querschnitt des Modells angezeigt werden;
- Lösungsgrafik entlang einer angegebenen Linie oder Fläche;
- Jede der o.g. Darstellungen kann im Zeitverlauf betrachtet und evtl. in einem Bild- oder Daten-File gespeichert werden.

6.1.2 Zeitadaptivität

Obwohl die verwendeten Einschritt-θ-Schemata unbedingt stabil sind (Abschnitt 3.1.1), spiegelt sich die Wahl eines zu großen Zeitschrittes insbesondere bei schnell ändernden Strömungen (wie z.B. eine Zylinderumströmung, Abschnitt 6.3.1) in einer sehr schlecht konditionierten Steifigkeitsmatrix, die viele präkonditionierte Iterationen zur Lösung benötigt, wider. Da jede neue FGMRES-Iteration langsamer wird und zusätzlichen Speicher braucht, war es sinnvoll, die Iterationszahl auf 100 zu beschränken und eine Zeitadaption bezüglich des FGMRES-Residuums einzuführen. Wenn also eine vorgegebene Toleranz (z.B. 10^{-5}) innerhalb 100 Iterationen nicht erreicht ist, wird der Zeitschritt halbiert. Umgekehrt, wenn das FGMRES-Verfahren weniger als 10 Iterationen gelaufen ist, kann der Zeitschritt verdoppelt werden. Letzteres tritt häufig dann ein, wenn sich die Strömung einem stabilen Zustand nähert.

6.2 Vergleich mit analytischen Lösungen

6.2.1 Kanalströmung

Analytische Lösung

Wir betrachten eine zweidimensionale Strömung im Kanal der Länge 20 mit dem Radius bzw. dem halben Querschnitt $R = 1$. Die Kanalachse falle mit der x-Achse zusammen, y sei die vertikale Koordinate. Die Geschwindigkeitskomponente in vertikaler Richtung ist null. Die Geschwindigkeitskomponente in x-Richtung werde mit u bezeichnet, sie ist nur von y abhängig. Der Druck in jedem Querschnitt ist konstant. Damit bleibt von den Navier-Stokes-Gleichungen lediglich die Gleichung für die x-Richtung übrig, die sich auf

$$\eta \left(\frac{d^2 u}{dy^2} + \frac{1}{y} \frac{du}{dy} \right) = \frac{dp}{dx} \tag{6.1}$$

reduziert, wobei am Rand des Kanals die Haftbedingung $u = 0$ gilt. Nach der Lösung der Gleichung (6.1) bildet sich ein parabolisches Geschwindigkeitsprofil aus

$$u(y) = u_{max}\left(1 - \frac{y^2}{R^2}\right), \qquad u_{max} = 2\bar{u} = \frac{R^2}{4\eta}\left(-\frac{dp}{dx}\right). \tag{6.2}$$

Der Zusammenhang zwischen Druckgefälle und mittlerer Geschwindigkeit $\bar{u}$ wird durch die dimensionslose Reibungszahl λ ausgedrückt, die durch folgende Beziehung definiert ist:

$$-\frac{dp}{dx} = \frac{\lambda\rho}{4R}\bar{u}^2, \qquad \bar{u} = \frac{q}{2R}, \tag{6.3}$$

wobei q der Massenstrom ist. Aus (6.2) folgt damit

$$\lambda = \frac{64}{\mathrm{Re}}, \qquad \mathrm{Re} = \frac{2\bar{u}R}{\nu} = \frac{2\bar{u}\rho R}{\eta}. \tag{6.4}$$

Die Substitution ergibt die endgültige Lösung:

$$u(y) = \frac{q}{\rho R^3}(R^2 - y^2). \tag{6.5}$$

Diese Beziehung befindet sich in ausgezeichneter Übereinstimmung mit Messungen, solange die Reynoldszahl unterhalb des kritischen Niveaus $\mathrm{Re}_{krit} = 2300$ liegt. Oberhalb dieses Wertes wird die Rohrströmung turbulent (Abb. 6.4). Weitere analytische Lösungen sowohl laminarer als auch turbulenter Strömungen unter verschiedenen Bedingungen findet man in [150], Kap. 5.

Numerische Tests

Für die numerischen Experimente im Kanal wurden zwei unterschiedliche Gitter erzeugt, auf denen die Navier-Stokes-Gleichungen mit den gemischten LBB-stabilen Stokes-Elementen diskretisiert wurden.

1. Ein strukturiertes Viereckgitter 100×10 enthält 1000 Zellen und 1111 Ecken (Abb. 6.2a). Die Taylor-Hood-Diskretisierung Q_2/Q_1 liefert 8442 Geschwindigkeits- und 1111 Druck-Freiheitsgrade; das Q_2/Q_0-Element - 8442/1000.
2. Ein unstrukturiertes Dreieckgitter mit ebenfalls 100 und 10 Punkten entlang x- und y-Achsen enthält 2206 Zellen und 1214 Ecken (Abb. 6.2b). Die Taylor-Hood-Diskretisierung P_2/P_1 liefert 9266 Geschwindigkeits- und 1214 Druck-Freiheitsgrade; das P_2/P_0-Element - 9266/2206.

Die Flüssigkeit wurde ähnlich zu einem Maschinenöl mit den Eigenschaften $\rho = 1000\frac{kg}{m^3}$ und $\eta = 0.088\frac{kg}{ms}$ gewählt. An den Rändern des Kanals ist die Haftbedingung $u = 0$, $v = 0$ gesetzt. Die Einströmung ist durch die mittlere Geschwindigkeit $\bar{u} = 0.01\frac{m}{s}$ definiert, das dem Massenstrom $q = 0.02\frac{m^3}{s}$ und der Reynoldszahl $\mathrm{Re} = 227$ entspricht.

In beiden Fällen liegen die numerischen Lösungen ganz nahe bei dem analytisch berechneten parabolischen Strömungsprofil (6.5) sowie den Ergebnissen des FLUENT-Simulationsprogramms (Abb. 6.3), wobei bei FLUENT ein doppel so feines strukturiertes Viereckgitter benutzt wurde. Damit wurde erreicht, dass FLUENT

a)

b)

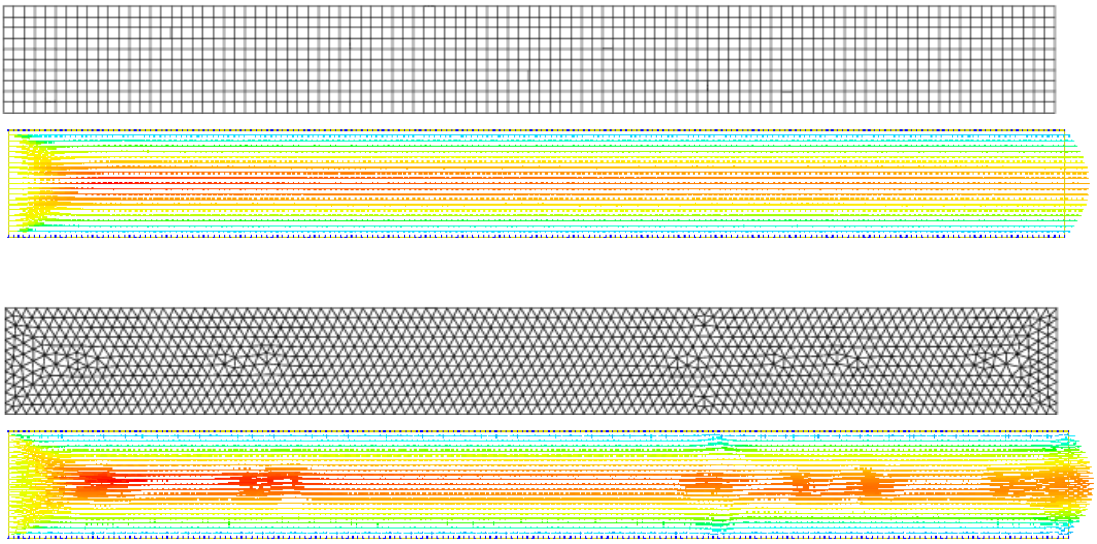
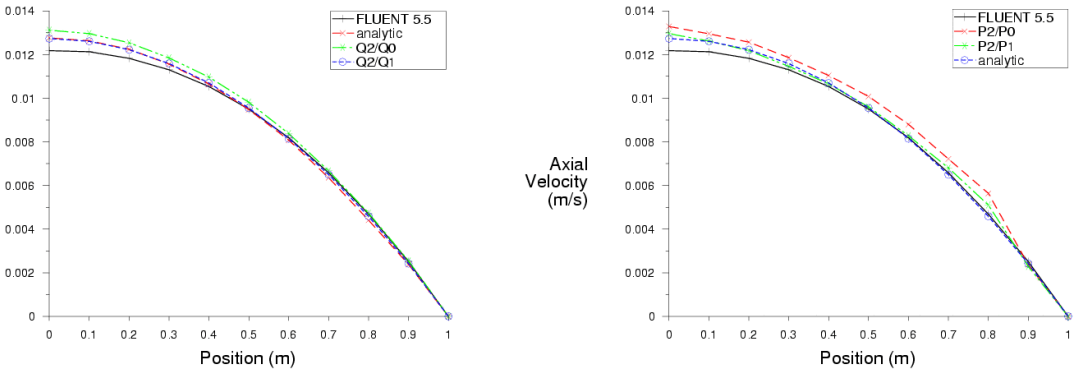


Abbildung 6.2: Laminare Strömung im Kanal diskretisiert mit einem strukturierten Viereck- und einem unstrukturierten Dreieckgitter



a) Strukturiertes Viereckgitter b) Unstrukturiertes Dreieckgitter

Abbildung 6.3: Kanalströmungsprofil: Vegleich zur analytischen Lösung

mit einer stabilisierten FVM erster Ordnung über die gleiche Anzahl der Freiheitsgrade gerechnet hat, wie sie ein Q_2/Q_1-Element erzeugt. Trotz dieser Anpassung konvergierte die FLUENT-Lösung sehr langsam zum erwarteten parabolischen Profil und benötigte deshalb einen längeren Kanal bei gleichen Fluideigenschaften und Randbedingungen. Die dargestellten Tests (Abb. 6.2,a,b) zeigen dagegen eine schnelle Einstellung eines konstanten Profils. Man kann allerdings einige Verdichtungen in der Geschwindigkeitsvektor-Darstellung auf dem unstrukturierten Gitter (Abb. 6.2,b) erkennen, die auch zu kleinen Störungen auf dem Profil (Abb. 6.3,b) führen.

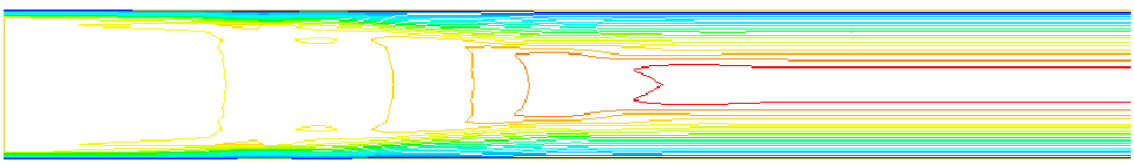


Abbildung 6.4: Kanalströmung: Turbulente Welle bei Re = 11364

Um den in der Theorie beschriebenen Übergang zur Turbulenz bei großen Reynoldszahlen zu simulieren, wurde die Einströmgeschwindigkeit auf $\bar{u} = 0.5\frac{m}{s}$ erhöht. Das entspricht einem Massenstrom $q = 1\frac{m^3}{s}$ und der Reynoldszahl Re = 11364. Dabei verbreitet sich im Kanal eine turbulente Welle (Abb. 6.4), wobei die Strömungsgeschwindigkeit im Profil keine parabolische Form mehr hat, sondern etwas größer an den Seiten im Vergleich zur Mitte des Kanals ist.

6.3 Umströmung eines Hindernisses

6.3.1 Zylinderumströmung in 2D

In einem ähnlichen zweidimensionalen Kanal wird ein Hindernis in der Form eines Kreises eingebaut, um den zeitlichen Ablauf einer instationären Strömung simulieren und untersuchen zu können. So ergibt sich ein bekanntes Benchmark-Modell, auf dem die instationären inkompressiblen Navier-Stokes-Gleichungen gelöst werden. Die Geometrie des Modells (Abb. 6.5) wurde aus [149] übernommen, wobei $R = 0.205$ und die Reynoldszahl durch die Beziehung (6.4) bestimmt wird. Man kann dadurch die Simulationsresultate mit bestimmten Fluideigenschaften und erwartetem Strömungsverhalten prüfen. Die Strömung wird mit den Parametern $\rho = 1$ und $\eta = 0.001$ untersucht. Wir interessieren uns insbesondere für die instationäre Strömung, die viel mehr Aufwand bzw. Rechenzeit bei der numerischen Lösung benötigt. Dabei kann das Konvergenzverhalten der iterativen Gleichungssystem-Solver untersucht und die Präkonditionierungs-Parameter abgestimmt werden. Der instationäre Fall entspricht der durchschnittlichen Einströmgeschwindigkeit $\bar{u} = 1.5\frac{m}{s}$ und der Reynoldszahl Re = 100, wobei die Strömung bei $\bar{u} = 1\frac{m}{s}$ (laut den Benchmark-Tests [149]) schon stationär ist.

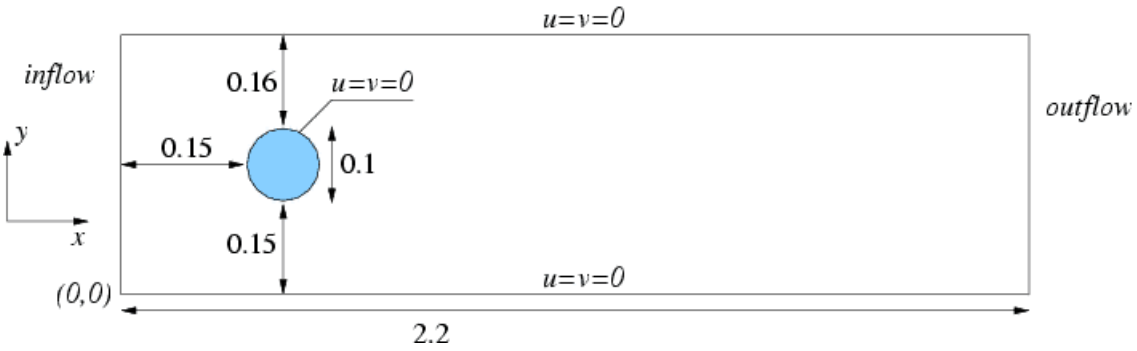


Abbildung 6.5: Kanal mit Zylinder

Die Tests werden auf zwei verschiedenen Gittern ausgeführt:

1. Ein unstrukturiertes Viereckgitter mit 1252 Zellen und 1352 Knoten (Abb. 6.6a),
   - Q_2/Q_0-Element: 10416/1252 Freiheitsgrade,
   - Q_2/Q_1-Element: 10416/1352 Freiheitsgrade,
2. Ein unstrukturiertes Dreieckgitter mit 1284 Zellen und 710 Knoten (Abb. 6.6b),

- P_2/P_0-Element: 5408/1284 Freiheitsgrade,
- P_2/P_1-Element: 5408/710 Freiheitsgrade.

Als Zeitdiskretisierungs-/Linearisierungs-Verfahren werden gewählt: Euler/Semi-Explizit, Euler/Newton, Crank-Nicolson/Semi-Explizit und Crank-Nicolson/Newton. Die Anwendung unterschiedlicher Zeitdiskretisierungs-Verfahren erster (Euler) und zweiter (Crank-Nicolson) Ordnung aus der Einschritt-θ-Familie führt zu einer zusätzlichen Untersuchung der Verfahren auf Sensitivität mit den instationären Strömungen nahe dem stationären Fall.

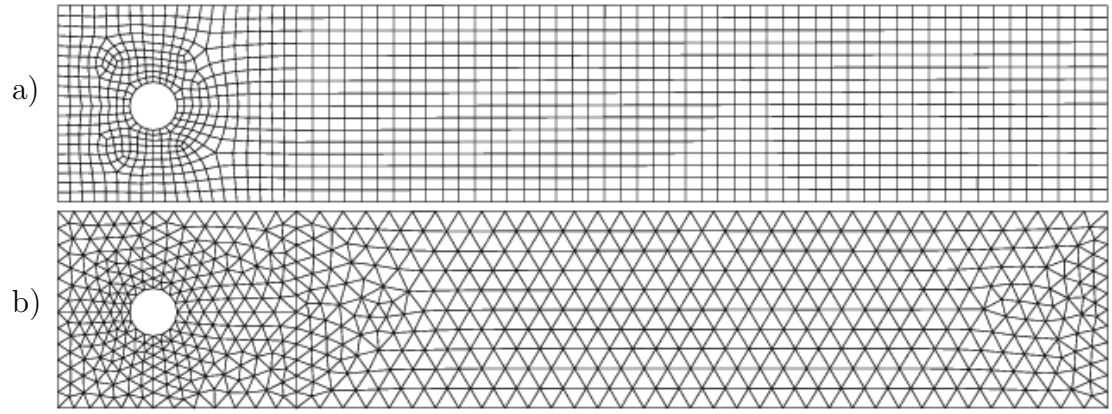


Abbildung 6.6: Unstrukturierte Viereck- und Dreieck-Gitter zur Zylinderumströmungs-Simulation

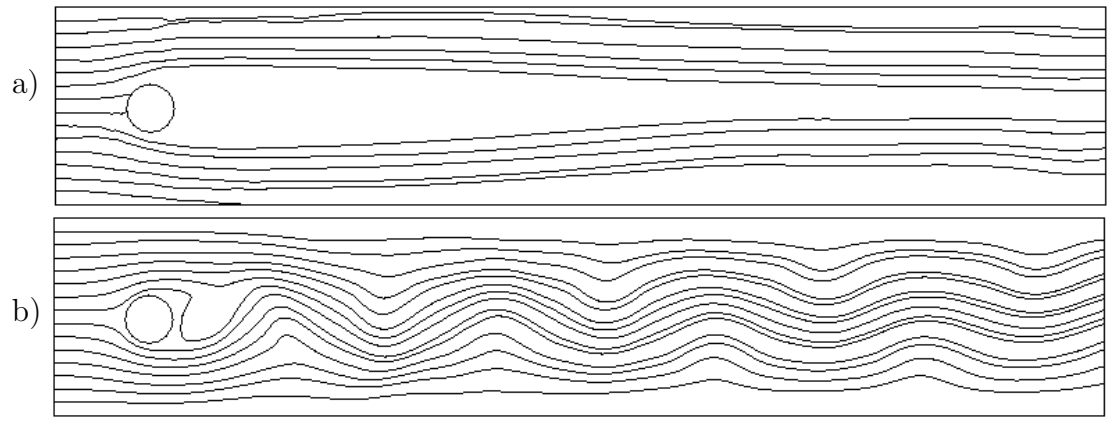


Abbildung 6.7: Stromlinien der Zylinderumströmung, diskretisiert mit dem Crank-Nicolson/Semi-Explizit-Schema: a) P_2/P_0-, b) P_2/P_1-Stokes-Element; $t = 3.0s$ in beiden Fällen

Alle Simulationen wurden über einen Zeitraum von 3 bis 5 Sekunden ausgeführt, so dass die Strömung zu einem periodischen Zustand gelangt. Das Strömungsverhalten bei den P_2/P_0- und P_2/P_1-Elementen und allen Zeitdiskretisierungen ist sehr unterschiedlich, wobei das P_2/P_1-Element ein viel stärkeres Wirbelbild wiedergibt (Abb. 6.7 und 6.8, A.5 und A.6). Die Strömung beim P_2/P_0-Element stabilisiert sich dagegen und weist nur leichte lange Wellen auf (Abb. 6.7a, 6.8a). Der Unterschied zwischen den beiden Modellen, diskretisiert jeweils mit dem Crank-Nicolson/Semi-Explizit- und dem Euler/Newton-Schema, liegt in der starken Welle nach dem Start

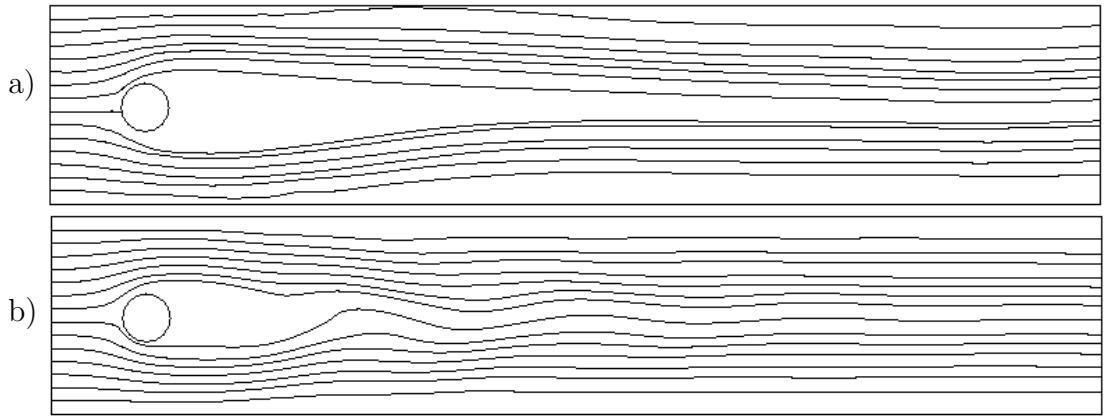


Abbildung 6.8: Stromlinien der Zylinderumströmung, diskretisiert mit dem Euler/Newton-Schema: a) P_2/P_0, $t = 1.16s$; b) P_2/P_1, $t = 3.0s$

im ersten Fall. Die Stabilisierung des Strömungsverhaltens wird auch dadurch erwiesen, dass der Zeitschritt durch das adaptive Verfahren (Abschnitt 6.1.2) größer wird. Den gleichen Unterschied gibt es bei den Crank-Nicolson/Newton- und dem Euler/Semi-Explizit-Schema (auch mit P_2/P_0-Element), dass die in der Zeitdiskretisierung liegende Ursache des Effekts erklärt.

Alle Simulationen mit dem Taylor-Hood-Element P_2/P_1 zeigen eine stark instationäre Strömung, die sich aber je nach Zeitschema und Linearisierung nach der Wellenlänge bzw. die Periode unterscheiden (Abb. 6.7b, 6.8b, 6.9a). Auf einem Konturenbild ist die Periode durch die Länge des Wirbelschwanzes hinter der Kugel gut zu merken und mit den anderen Bildern zu vergleichen (Abb. A.5b, A.6b). Dieser Wirbelschwanz oszilliert und reißt die wegfliegenden Wirbel von der Umströmung ab, die zusammen die sog. Taylorsche Wirbelstraße bilden. Die stärksten Wellen bzw. die kürzeste Periode treten beim Crank-Nicolson/Semi-Explizit-Schema (Abb. 6.7b), die mittelstarken Wellen sind bei den Crank-Nicolson/Newton- (Abb. 6.9a) und Euler/Semi-Explizit-Schemata und die längste Periode - bei dem Euler/Newton-Schema (Abb. 6.8b) auf.

Das Beispiel mit dem Crank-Nicolson/Newton-Schema und dem Taylor-Hood-Element P_2/P_1 betrachten wir genauer in der Zeitentwicklung. Dafür wurden zwei Querschnitte parallel zur y-Achse definiert (Abb. 6.9a):

1. $x = 0.4$, kurz nach dem Zylinder,
2. $x = 1$, kurz vor der Mitte des Kanals.

Über diesen Querschnitten wurden Grafiken des Geschwindigkeits-Betrages dargestellt und im zeitlichen Verlauf beobachtet (Abb. 6.9,b,c). Die erste Kurve entspricht der Zeit $t = 0.05s$ und stellt ein symmetrisches Strömungsprofil dar. Weitere Kurven wurden nach der Einstellung der konstanten Periode $t_p \approx 0.3s$ genommen, um das Verhalten innerhalb einer Periode zu veranschaulichen. Maximalwerte der Geschwindigkeits-Betrages bewegen sich links und rechts entlang des Querschnittes, wobei beim ersten Querschnitt noch ein durch den Zylinder initiiertes lokales Minimum liegt. In der Abbilding 6.9,b,c sind die letzte linke, die letzte rechte und die zwei mittleren Positionen in der Kurvenbewegung dargestellt. Die linke bzw. die

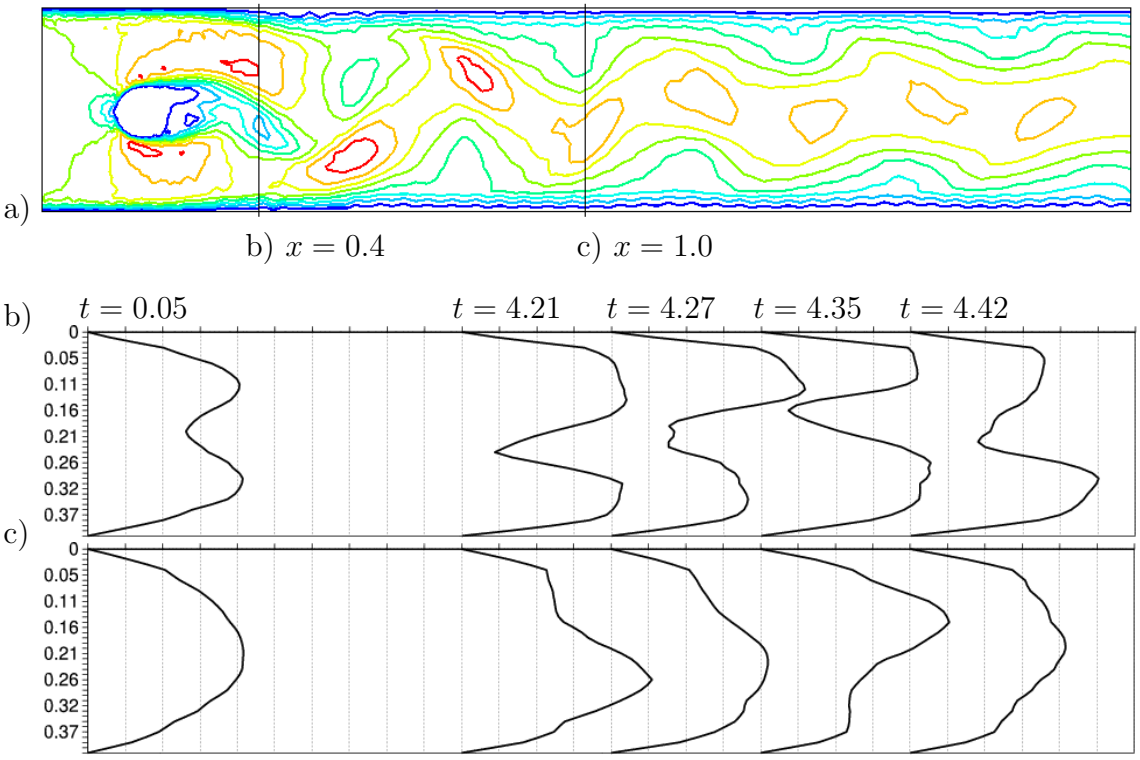


Abbildung 6.9: Geschwindigkeits-Konturen der Zylinderumströmung, diskretisiert mit dem Crank-Nicolson/Newton-Schema und P_2/P_1-Stokes-Element, $t = 3.6s$

rechte Kurvenposition auf der Abbildung 6.9b entspricht ungefähr der mittleren ausgeglichenen Position auf der Abbildung 6.9c und umgekehrt, was durch eine andere Auswahl der Querschnitte geändert werden könnte.

Die gleichen Simulationen auf einem unstrukturierten Viereckgitter sind misslungen, dadurch haben sie aber eine wesentliche Erkenntnis gebracht. Durch stark verformte Vierecke in der Nähe des Kreises sind große Geschwindigkeitsgradienten an diesen Stellen schon nach einigen Zeitschritten gut erkennbar, obwohl das gesamte Strömungsbild zunächst dem normalen Verlauf entspricht (Abb. 6.10). Nach ca. $1.5s$ beginnen die Geschwindigkeiten in der Nähe des Kreises schnell zu wachsen und das Strömungsverhalten zu zerstören. Die Zeitadaption verkleinert dabei den Zeitschritt bis auf Computer-Genauigkeit. Einen Einsatz von feineren Gittern hat zwar zu einer Verzögerung der Zerstörung geführt, aber das gesamte Resultat nicht geändert.

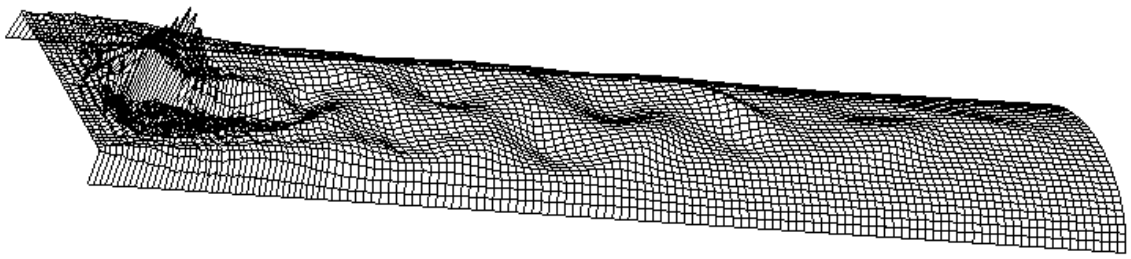


Abbildung 6.10: Geschwindigkeits-Beträge der Zylinderumströmung, diskretisiert mit dem Crank-Nicolson/Newton-Schema und Q_2/Q_1-Stokes-Element, $t = 1.5s$

6.3.2 Komplexe Umströmung in 3D

Eine andere Hindernisumströmung wurde im dreidimensionalen Fall simuliert. Die Modell-Geometrie (Abb. 6.11) wurde so konstruiert, dass gleichzeitig mehrere Verwirbelungen und deren Mischungen entstehen können. In der xz-Ebene ist das eine symmetrische Umströmung des Quaders, obwohl das rechteckige Hindernis die obere Schranke des Modells nicht berührt. In der xy-Ebene entsteht dadurch eine Umströmung des Hindernisses von oben und weiter eine Strömung über eine Stufe. Die vollständig rechteckige Geometrie wurde mit einem strukturierten kubischen Gitter mit dem Parameter $h = 0.05$ diskretisiert. Das Gitter enthält 20544 Zellen und 23373 Knoten, das bei einer Geschwindigkeitsapproximation zweiter Ordnung (Q_2) 526443 Geschwindigkeits-Freiheitsgrade ergibt.

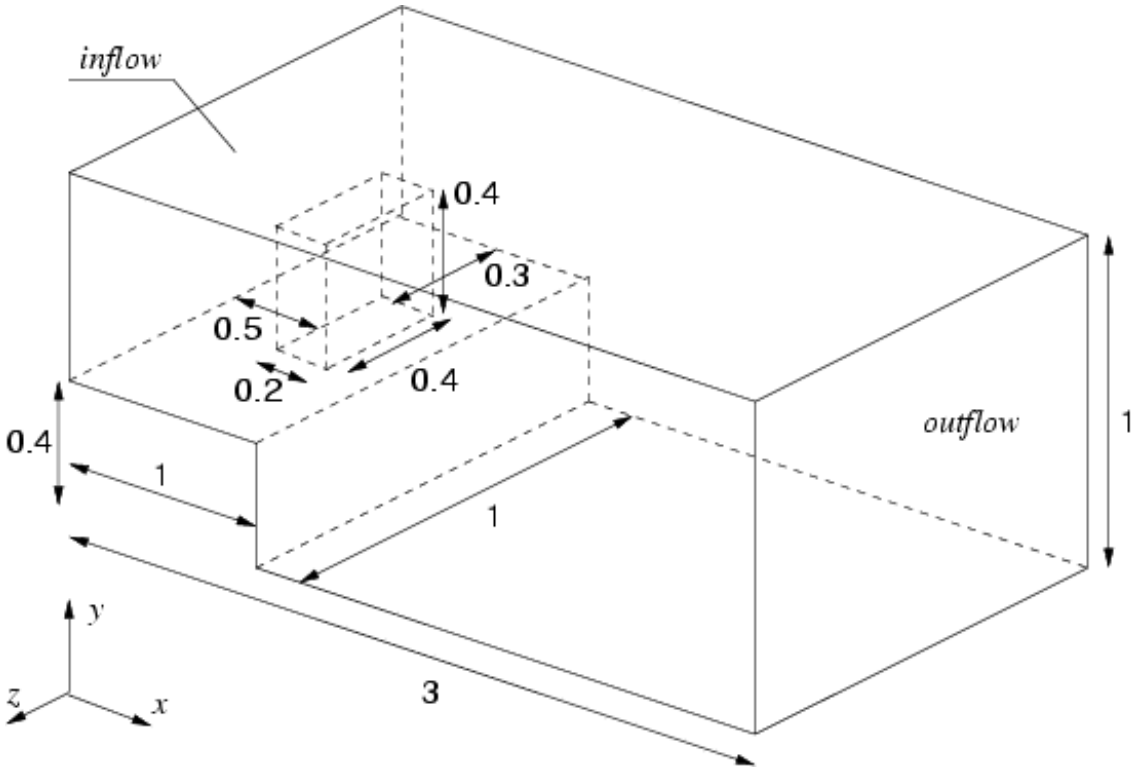


Abbildung 6.11: Dreidimensionale Modell-Geometrie mit einem Hindernis

Als Strömungsparameter wurden die gleichen wie im vorhergehenden Abschnitt gewählt: $\rho = 1$, $\eta = 0.001$, wobei die Einströmgeschwindigkeit etwas niedriger $\bar{u} = 0.5\frac{m}{s}$ gewählt wurde. Für das Modell wurde das Euler/Semi-Explizit-Schema gewählt, das sich mit dem zweiter Ordnung Crank-Nicolson/Newton-Schema vom letzten Abschnitt gut vergleichen ließ. Das Euler/Semi-Explizit-Schema erzeugt eine weniger besetzte globale Matrix, was zu einer merklichen Einsparung an Speicherplatz führt und die Rechenzeit reduzieren kann. Im Raum wurde das Modell mit den Q_2/Q_0- und Q_2/Q_1-Elementen diskretisiert.

Die Strömung entwickelt sich wie erwartet. Links, rechts und über dem Hindernis steigt die Strömungsgeschwindigkeit und initiiert sofort einen abreißenden Wirbel hinter dem Hindernis (Abb. 6.12). Etwas später entstehen zwei symmetrische Wirbel im Schatten der Stufe (Abb. 6.13). Die drei Strömungen mit der erhöhten Geschwin-

digkeit und Wirbeln nähern und vereinigen sich am Ausstrom (Abb. A.8, A.9).

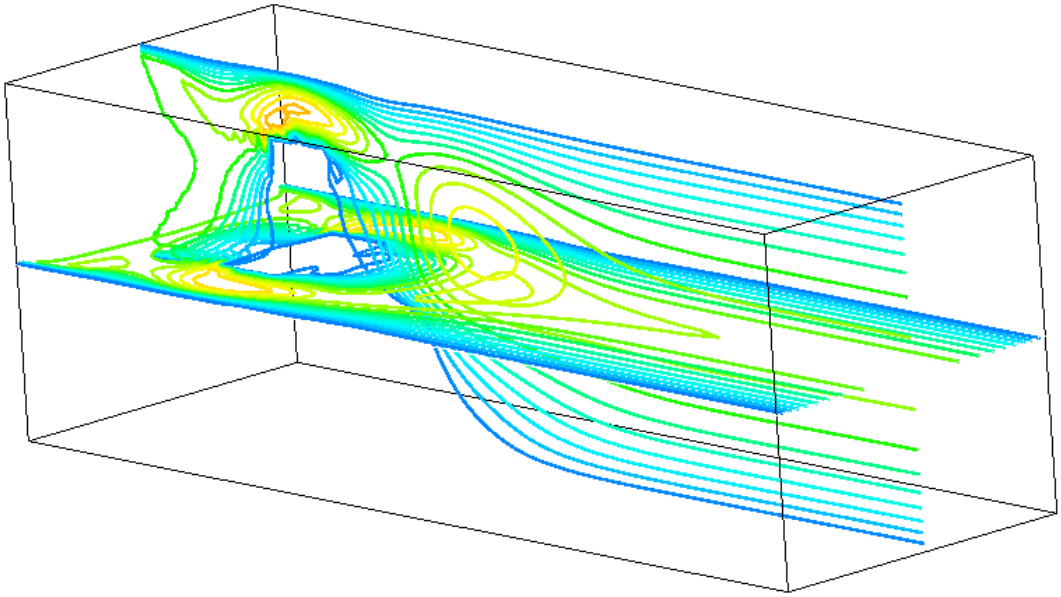
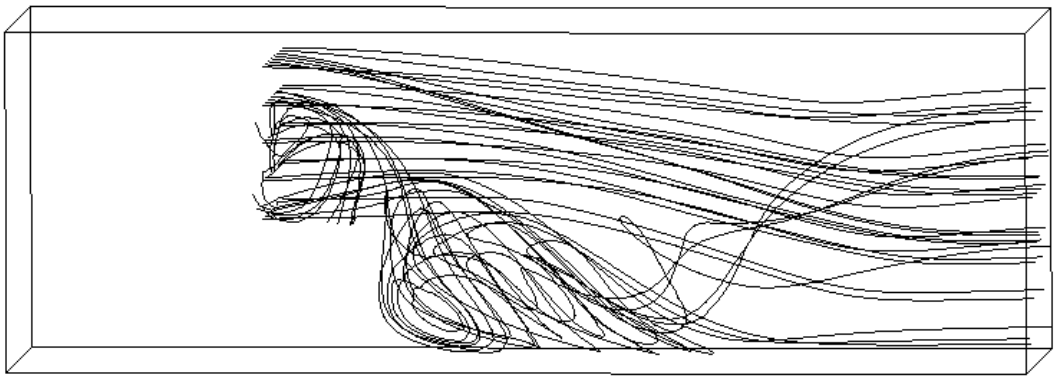
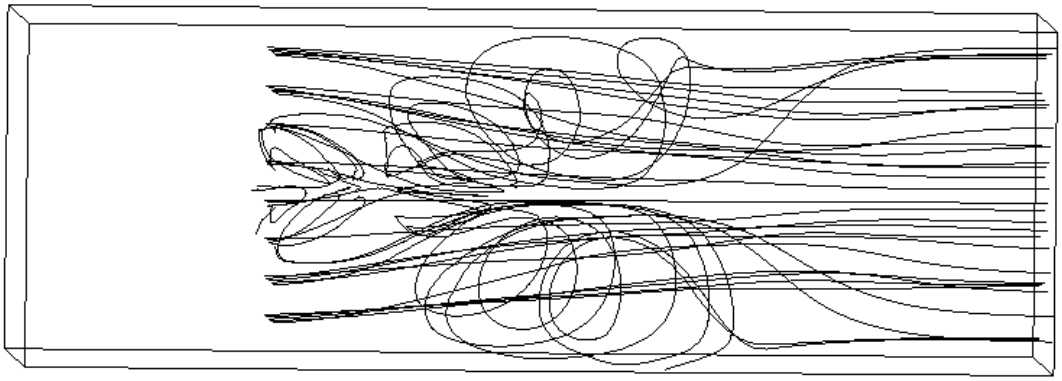


Abbildung 6.12: Umströmung eines Hindernisses in 3D: Konturen des Geschwindigkeits-Betrages, Q_2/Q_1-Element, $t = 0.25s$



a) Blick auf die xy-Ebene (von der Seite)



b) Blick auf die xz-Ebene (von oben)

Abbildung 6.13: Umströmung eines Hindernisses in 3D: Stromlinien, beginnend vom yz-Querschnitt mit $x = 0.75$, Q_2/Q_1-Element, $t = 2.25s$

6.4 Verfahren höherer Ordnung in 2D

Die konformen finiten Elementen werden im Simulations-Solver durch Eingabe des Geometrie-Typs der Aufgaben-Dimension und der Ordnung der polynomialen Ansatzfunktionen definiert (siehe das Merkmaldiagramm 5.7 und die Implementierung im Abschnitt 4.4). Ein gemischtes Stokes-Element besteht aus zwei solchen konformen finiten Elementen gleicher Geometrie und Dimension. Deshalb können theoretisch beliebige Kombinationen P_i/P_j und Q_i/Q_j im Solver benutzt werden, wobei wir die Menge der LBB-stabilen Elemente daraus verwenden. In der Praxis ist die Anwendung von Elementen höherer Ordnung durch die vorhandene Rechenleistung begrenzt, weil mehrere lokale Matrizen für ein Stokes-Element erzeugt werden müssen, wobei jeder Matrixeintrag eine Multiplikation zweier polynomialer Ansatzfunktionen und eine nachfolgende exakte Integration des Produktes erfordert (Abschnitt 4.3). Eine naive Implementierung des Polynomproduktes besitzt eine polynomiale Komplexität und erreicht sehr schnell die Rechenleistungs-Grenze. So konnten maximal Q_6/Q_5- bzw. P_7/P_6-Elementmatrizen im zweidimensionalen Fall und Q_4/Q_3- bzw. P_5/P_4-Elementmatrizen für den dreidimensionalen Fall berechnet werden. Einen Einsatz der modernen algebraischen Algorithmen können diese Grenze bis 10.-12. Ordnung verschieben, was eine mögliche Weiterentwicklung des Solvers wäre.

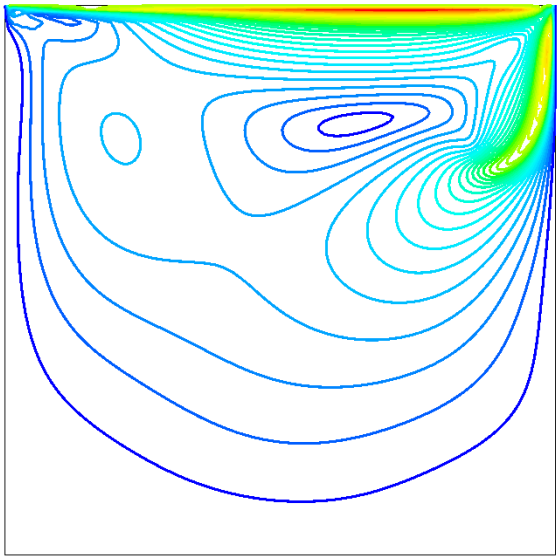
Zum Testen der Ansätze höherer Approximationsordnung (> 2) wurde ein gut bekanntes und untersuchtes, aber trotzdem interessantes Modell einer Nischenströmung (engl. driven cavity) in einem quadratischen und einem Dreieck-Gebiet gewählt. Auf dem oberen Rand des Gebietes ist die tangentiale Geschwindigkeit $\bar{u} = 0.1\frac{m}{s}$ angegeben, wobei auf anderen Rändern die Haftbedingung $\boldsymbol{u} = 0$ definiert ist. Durch die tangentiale Geschwindigkeit wird die Flüssigkeit in eine Rotationsbewegung gebracht. Die Flüssigkeit ist ölähnlich mit den Parametern $\rho = 1000$ und $\eta = 0.1$. Zur Zeitdiskretisierung/Linearisierung wurde das Crank-Nicolson/Newton-Schema für alle Tests ausgewählt.

6.4.1 Viereckgitter

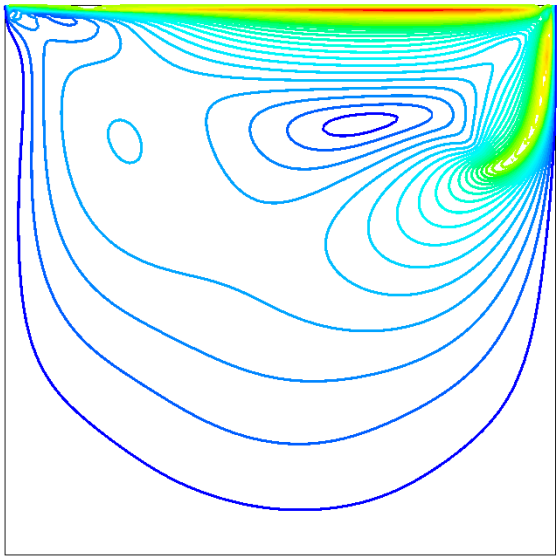
Das quadratische Gebiet der Einheitsgröße wurde im Zusammenhang mit der Nischenströmung am häufigsten in der Literatur erwähnt (siehe z.B. [73], Kap. 5), so dass eine günstige Gelegenheit zum Vergleich der Ergebnisse gegeben ist. Um darüber hinaus die p- mit der h-Modellverfeinerung vergleichen zu können, wurden zwei unterschiedliche Gitter: $\mathcal{T}_1$ mit $h = \frac{1}{32}$ und $\mathcal{T}_2$ mit $h = \frac{1}{64}$ erzeugt, so dass die numerische Lösung mit den jeweils Q_4/Q_3- und Q_2/Q_1-Taylor-Hood-Elementen die gleiche Anzahl der Geschwindigkeits-Freiheitsgrade besitzt.

Die rotierende Strömung des viskosen Öls entwickelt sich ziemlich langsam, da die Viskosität relativ hoch und die Geschwindigkeit $\bar{u}$ relativ klein sind. Die entsprechende Reynoldszahl $\mathrm{Re} = 1000$ verspricht aber die Entstehung von zwei bis drei Wirbeln [73]. Dabei spielt die adaptive Zeitschrittänderung (Abschnitt 6.1.2) eine wichtige Rolle. Der Anfangswert $\triangle t = 0.01$ vergrößert sich mehrfach am Anfang und nimmt die Werte 0.16 oder 0.32 im Laufen der Simulation an. Das gestattet eine schnelle Durchführung von mehreren Berechnungen mit einem großen Zeitinterval, das in dieser Untersuchung notwendig ist.

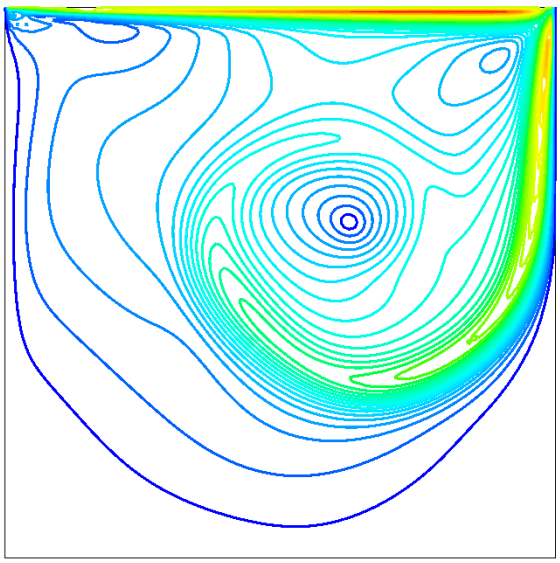
Das Strömungsbild wird in zwei unterschiedlichen Zeitpunkten $t \approx 5$ und $t \approx 25$ mit Hilfe der Geschwindigkeitskonturen verglichen. Obwohl eine exakte Übereinstimmung der Zeitpunkte wegen der Adaptivität nicht erreichbar ist, sehen die Bilder zu den entsprechenden Zeitpunkten (Abb. 6.14a,b und c,d) sehr ähnlich aus. Eine starke Strömung läuft vom oberen Modellrand entlang der rechten Seite und erzeugt in der oberen rechten Ecke einen starken Wirbel, der sich mit der Zeit über das gesamte Strömungsgebiet verbreitet. In den Ecken entstehen dabei kleinere Sekundärwirbel. Dieses Verhalten entspricht den Berechnungen an demselben Modell mit der gleichen Reynoldszahl in [73].



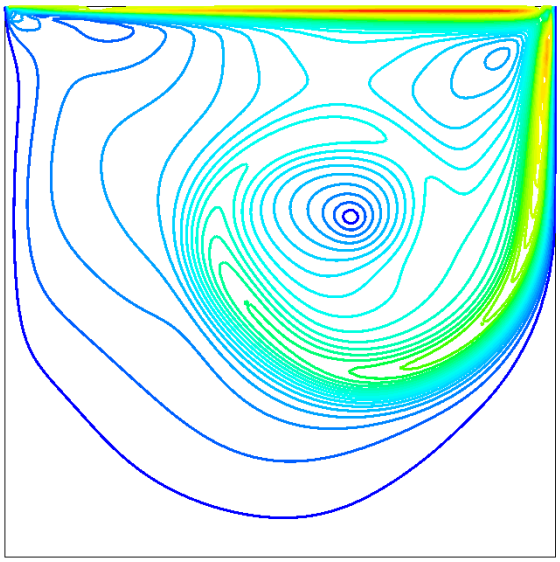
a) $\mathcal{T}_1,\ Q_4/Q_3,\ t = 5.04$



b) $\mathcal{T}_2,\ Q_2/Q_1,\ t = 5.12$



c) $\mathcal{T}_1,\ Q_4/Q_3,\ t = 25.06$

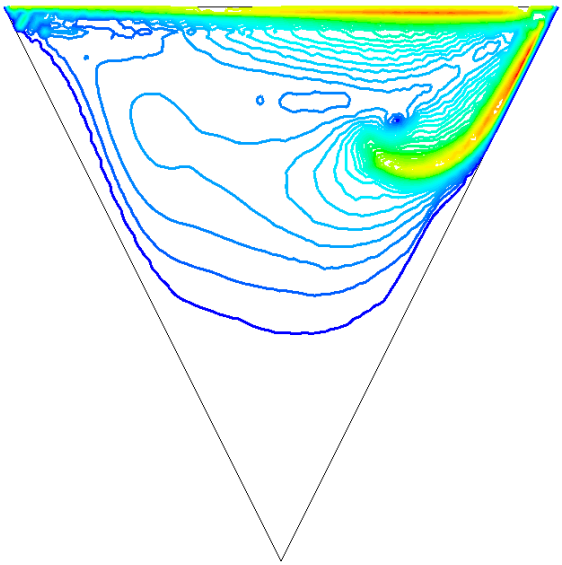


d) $\mathcal{T}_2,\ Q_2/Q_1,\ t = 25.14$

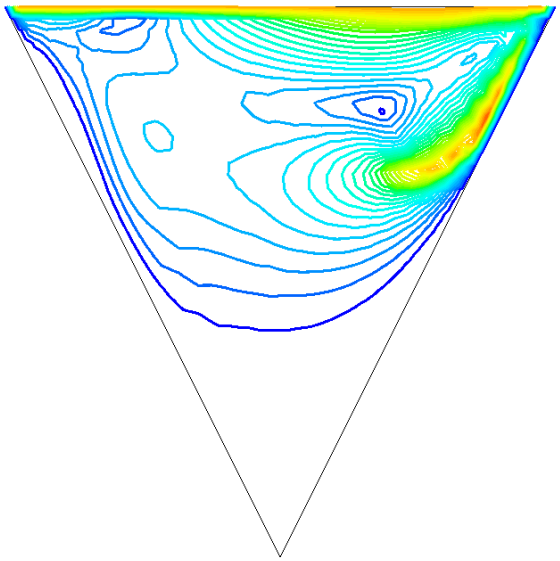
Abbildung 6.14: Nischenströmung auf strukturierten Viereck-Gittern mit Q_2/Q_1- und Q_4/Q_3-Elementen

6.4.2 Dreieckgitter

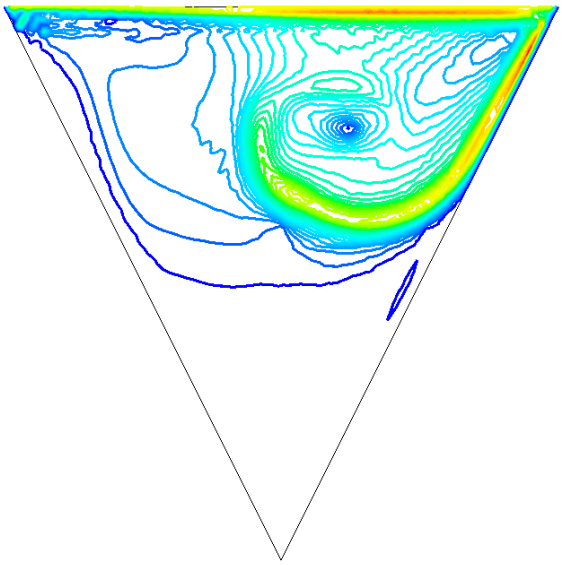
Analog zur Nischenströmung in einem Quadrat wird dieselbe Strömung in einem Dreieckgebiet betrachtet, um die Dreieckelemente höherer Ordnung austesten zu können. Die Breite des oberen Randes genau wie die Höhe des Dreieckgebietes hat die Länge 2.



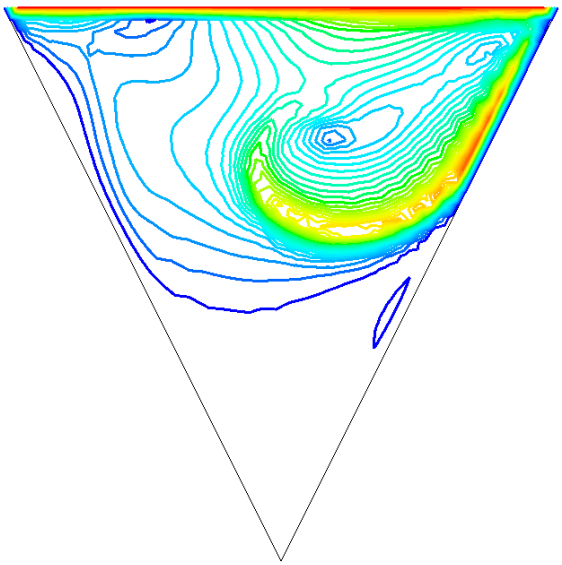
a) $\mathcal{T}_1,\ P_4/P_3,\ t = 14.81$



b) $\mathcal{T}_2,\ P_2/P_1,\ t = 14.73$



c) $\mathcal{T}_1,\ P_4/P_3,\ t = 30.18$



d) $\mathcal{T}_2,\ P_2/P_1,\ t = 30.11$

Abbildung 6.15: Nischenströmung auf unstrukturierten Dreieck-Gittern mit P_2/P_1- und P_4/P_3-Elementen

Das Gebiet wurde mit drei unstrukturierten Gittern und folgenden finiten Elementen diskretisiert:

$\mathcal{T}_1$: 452 Zellen und 259 Knoten:

- P_4/P_3-Element: 7490/2131 Geschwindigkeit-/Druck-Freiheitsgrade,

- P_5/P_4-Element: 11622/5910,

$\mathcal{T}_2$: 1852 Zellen und 992 Knoten:

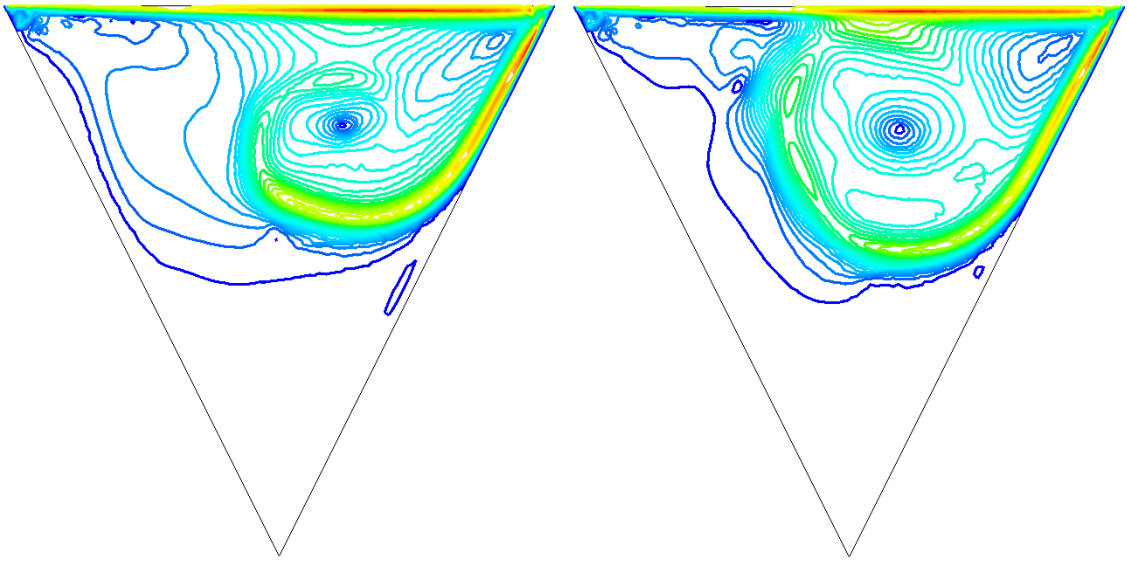
- P_2/P_1-Element: 7670/992, dass ähnlich dem P_4/P_3-Element auf $\mathcal{T}_1$ ist.

$\mathcal{T}_3$: 11494 Zellen und 5910 Knoten. Dieses hochaufgelöste Gitter wurde für eine Verifizierung der Resultate auf den vorhergehenden Gittern benutzt, da keine ähnlichen Modelle in der Literatur vorliegen.

- P_2/P_1-Element: 46626/5910,
- P_4/P_3-Element: 185202/52210.

Die Fluideigenschaften sowie die Randgeschwindigkeit $\bar{u}$ wurden nicht geändert. Durch die Geometrievergrößerung ist allerdings die Reynoldszahl auf $\mathrm{Re} = 2000$ gestiegen, was die Verwirbelung etwas erhöhen soll. Das gesamte Strömungsverhalten ist sehr ähnlich dem quadratischen Fall, obwohl der große Wirbel sich etwas langsamer bildet und wächst. Die Strömung im unteren Bereich ist sehr langsam und lässt nur selten einige Wirbel erkennen (z.B. Abb. 6.15c,d). Nur das höchstaufgelöste Modell mit dem Gitter $\mathcal{T}_3$ und P_4/P_3-Element zeigt Geschwindigkeits-Konturen im unteren Bereich auch in den ersten 10 Sekunden der Simulation (Abb. 6.17).

Auf der Abbildung 6.15 werden die Simulationen mit einer ähnlichen Anzahl der Geschwindigkeits-Freiheitsgrade zu den Zeitpunkten $t \approx 14.8$ und $t \approx 30.1$ verglichen. Ähnliche Resultaten weisen aber darauf hin, dass die höhere Approximationsordnung die Wirbel genauer abgebildet ist (Abb. 6.15c). Diese Berechnung hat aber dafür ungefähr das zwei- bis dreifache an CPU-Zeit gebraucht als auf dem Gitter $\mathcal{T}_2$ mit dem P_2/P_1-Element.

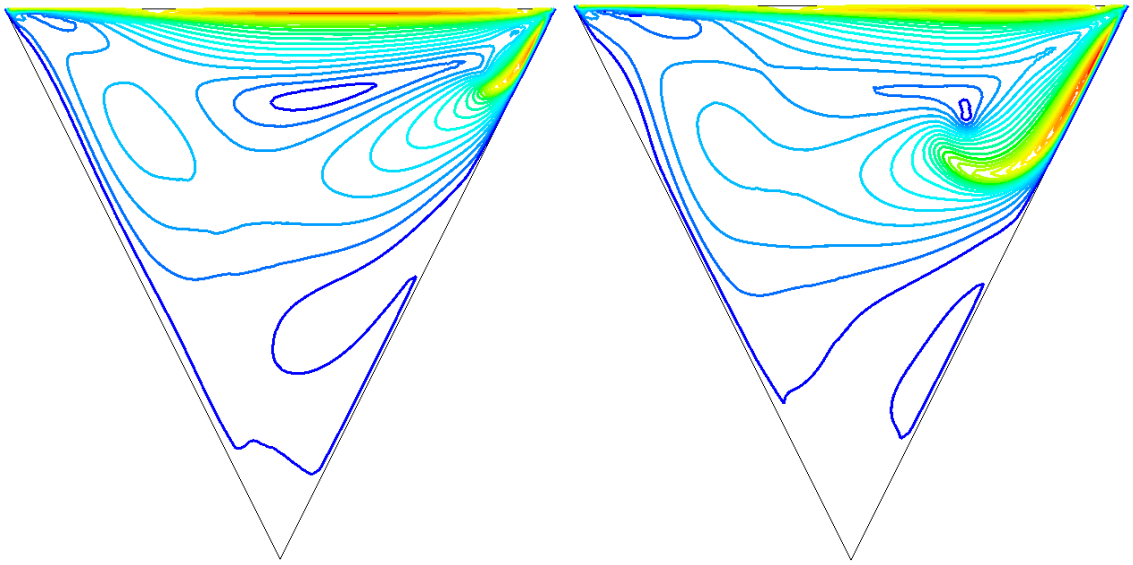


a) $t = 30.13$ b) $t = 50.00$

Abbildung 6.16: Nischenströmung auf dem unstrukturierten Dreieck-Gitter $\mathcal{T}_1$, diskretisiert mit dem P_5/P_4-Element

Eine Simulation auf dem feinen Gitter $\mathcal{T}_3$ mit dem P_4/P_3-Element ist in der Lage, wesentlich kleinere und langsamere Wirbel wiederzugeben. Dabei ist der Zeitschritt

adaptiv kleiner $\triangle t = 0.08$ eingestellt, wobei er bei anderen Tests zwischen 0.16 und 0.64 variiert.



a) $t = 5.80$ b) $t = 14.77$

Abbildung 6.17: Nischenströmung auf dem unstrukturierten Dreieck-Gitter $\mathcal{T}_3$, diskretisiert mit dem P_4/P_3-Element

6.5 Eine Pipeline in 3D

In diesem letzten Beispiel wird eine komplexe Pipeline betrachtet (Abb. 6.18). Das große Rohr der Länge 3m und Radius $R = 1m$ hat zwei kleinere symmetrisch platzierte Einström- (links im Bild) und zwei Ausström-Rohre (rechts im Bild) mit dem Radius $R_1 = 0.2$. Die Rohr-Paare sind um 1m von einander und vom großen Ein- und Ausström entfernt und mit 90 Grad bezüglich der Achse des großen Rohres zu einander gedreht. Das Modell wurde mit einem Tetraeder-Gitter diskretisiert, wobei Ein- und Ausström des großen Rohr 32 und der kleinen Rohren 12 Punkten auf dem Kreis haben. Das liefert ein ungleichmäßiges Gitter mit einer feineren Auflösung in den kleinen Rohren.

Im Gegensatz zu den vorhergehenden Tests wollen wir eine langsame Strömung mit Hilfe der Stokes-Gleichungen simulieren. Die Strömungsrichtung im großen Rohr ist von links nach rechts und durch die Einströmgeschwindigkeit $\bar{u} = 0.01\frac{m}{s}$ bestimmt. Diese langsame Strömung wird durch eine größere Einströmgeschwindigkeit $\hat{u} = 0.5\frac{m}{s}$ (gegeben senkrecht zur Einstomfläche) auf dem ersten Rohr-Paar und die gleiche Ausströmgeschwindigkeit auf dem zweiten Rohr-Paar gestört (Abb. 6.19). Die Fluideigenschaften $\rho = 1$ und $\eta = 0.1$ sind absichtlich so gewählt, dass sich eine kleine Reynoldszahl $\mathrm{Re} = 0.2$ ergibt und die Stokes-Gleichungen den Strömungsvorgang realistisch wiedergeben können.

Trotz mehrerer Ein- und Ausströmungen stabilisiert sich die Strömung in der Pipeline (Abb. A.10) im wesentlichen schon nach der ersten halben Sekunde der Simulation. Dieses Ergebnis wie auch das Strömungsbild werden durch die Tests auf einem feineren Gitter und durch Verfahren höherer Ordnung bestätigt.

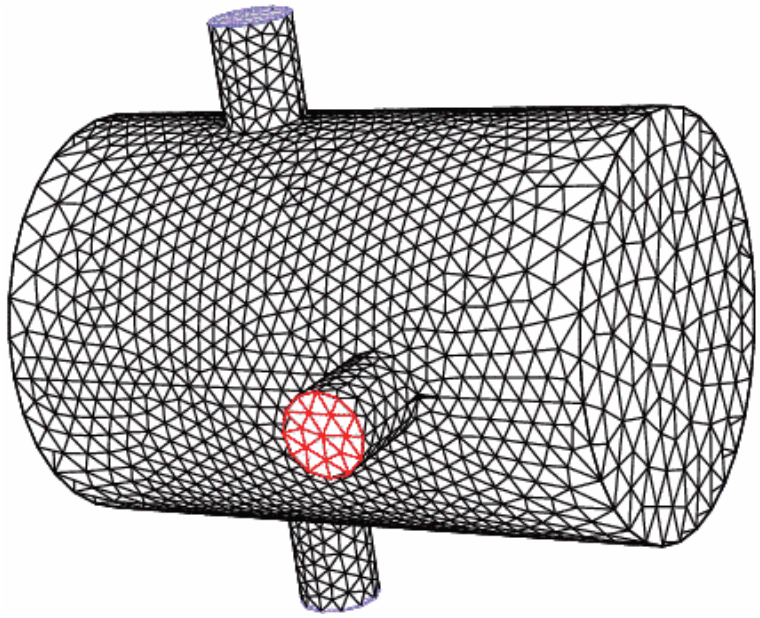


Abbildung 6.18: Dreidimensionale Modell-Geometrie und Tetraedergitter einer Pipeline

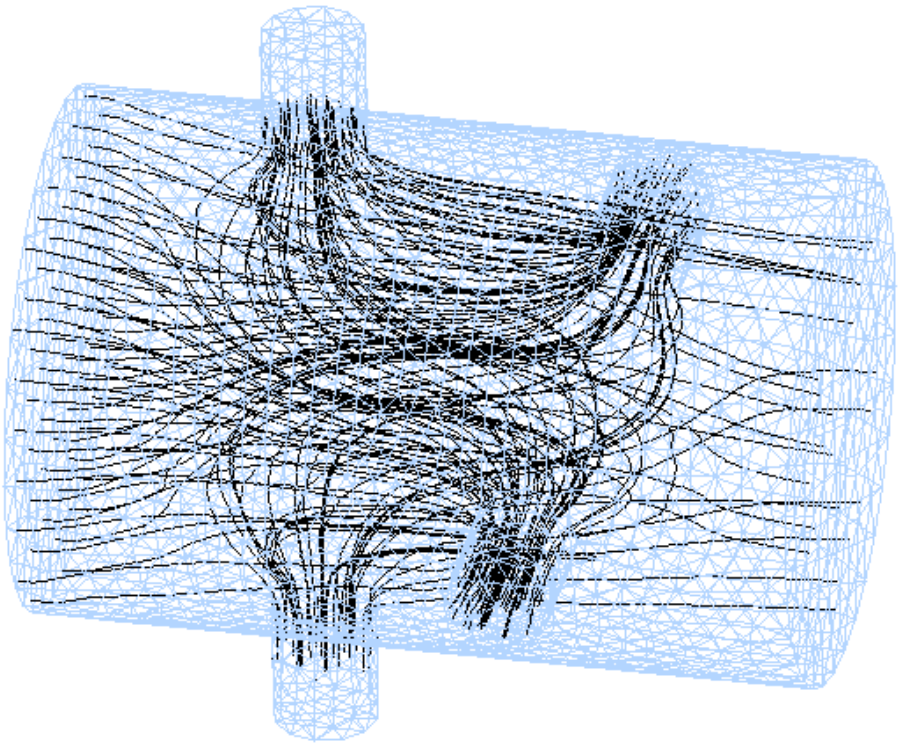


Abbildung 6.19: Stromlinien einer Stokes-Strömung in der Pipeline, diskretisiert mit Euler/Semi-Explizit-Schema und P_2/P_1-Element auf einem Tetraeder-Gitter

Kapitel 7

Zusammenfassung und Ausblick

So eine Arbeit wird eigentlich nie fertig, man muss sie für fertig erklären, wenn man nach Zeit und Umständen das möglichste getan hat.
- Johann Wolfgang von Goethe

In Rahmen dieser Arbeit wurde ein Strömungssimulations-Solver entwickelt, der die Navier-Stokes-Gleichungen oder deren Sonderfälle mit der FEM auf unstrukturierten Dreieck- oder Viereck-Gittern in ein-, zwei- und dreidimensionalen Fall lösen kann. Einige der implementierten FEM sind zu den konservativen FVM äquivalent (Abschnitt 3.5). Die FEM stellt aber eine größere Menge der Approximationsmöglichkeiten dar und wurde deshalb bevorzugt (Abschnitt 7.1).

Für die Softwareentwicklung wurde die Technologie der generativen Programmierung (GP) gewählt, die besondere Vorteile für wissenschaftliche Software öffnet (Abschnitt 7.3, 7.4). Im Problembereich der numerischen Strömungssimulation wurden viele mögliche moderne Verfahren berücksichtigt, die ein Simulations-Solver enthalten kann, was zu einem detaillierten und flexiblen Software-Modell bzw. Domänen-Modell (Abschnitt 7.2) geführt hat. Die auf der Grundlage dieses Modells entwickelte Software-Komponenten sind in hohem Maße wiederverwendbar und in unterschiedlichen Kombinationen bilden sie effiziente Simulations-Solver. Die Software eröffnet auch zahlreiche Wege der Weiterentwicklung (Abschnitt 7.5), die über die Grenzen des Problembereichs hinaus gehen können.

Die entwickelte Software wurde ***Generativer Simulations-Solver*** (GSS) genannt und im Internet zusammen mit der Entwickler- und Benutzer-Dokumentation veröffentlicht. Die Quelltexte sind frei zum Herunterladen verfügbar in der Form 'AS IS'. Die wichtigste und interessanteste Teile der Quelltexte wurden mit dem ***DoxyGen***-Tool kommentiert und damit in die Entwickler-Dokumentation mit eingeschlossen. Da Implementierung einiger Komponenten und zugehörige Dokumentation im Laufe der Zeit geändert werden können, bekommt man unter der u.g. Adresse immer die aktuelle Version. Diese Arbeit enthält allerdings diejenigen grundlegenden Konzepte und wichtigste Implementierungs-Information, die nicht geändert werden sollen.

`http://gss.scientificcpp.com`

7.1 Gesamtblick auf FDM, FVM und FEM

Die FDM, FVM und FEM haben verschiedene Ursprünge und Geschichte. Sie wurden von verschiedenen Arbeitsgruppen entwickelt, unterstützt bzw. bevorzugt. Deshalb besitzen sie unterschiedliche theoretische Grundlagen. In den aktuellen Untersuchungen dieser Verfahren versucht man einen Vergleich durchzuführen, um zuerst die Auswahl einer oder anderen Methode zu erklären. Die gefundene Ähnlichkeiten können nicht nur in der theoretischen Weiterentwicklung dieser Methoden mitwirken, sondern sind auch bei einer Programm-Umsetzung sehr wichtig.

Die FDM ist generell nur auf strukturierten Gittern anwendbar und ist in solchen Fällen auch effizienter bezüglich der Rechenzeit als eine allgemeine FVM oder FEM. Die letztgenannten Methoden nutzen die Integralform von Gleichungen, stammen aus der Methode der gewichteten Residuen und können deshalb in Einzellfällen zu den gleichen Steifigkeitsmatrizen führen (Abschnitt 3.5). Beide Methoden funktionieren auf allgemeinen unstrukturierten Gittern und können dadurch Aufgaben in komplexer Geometrien gut approximieren. Die Identifikation einer Methode (FVM oder FEM) bleibt aber unterschiedlich. Eine FVM wird durch die Randintegral-Approximationen definiert, wogegen eine FEM durch die Wahl der Ansatzfunktionen bestimmt wird. Es existieren nur wenige FVM höherer Approximationsordnung. Die FEM höherer Ordnung ist dagegen einfach durch polynomiale Ansatzfunktionen entsprechender Ordnung definiert. Man kann also behaupten, dass die FVM eine Untermenge der FEM ist (siehe auch [70], S.xxxvi). Da wir eine große Methoden-Anzahl mit einem kleinerem Aufwand umsetzen wollen, ist die FEM dafür viel günstiger geeignet. Darüber hinaus stehen hinter jeder FEM die vorher gespeicherten lokalen Matrizen (Abschnitt 5.2.3), die direkt zur globalen Steifigkeitsmatrix addiert werden, was wesentlich effizienter als eine Assemblierung durch Randintegral-Approximationen einer FVM ist.

7.2 Das entwickelte Domänen-Modell und die implementierten Komponenten

Das entwickelte Domänen-Modell kann natürlicherweise nicht vollständig sein. Neue Methoden gibt es schon heute oder werden morgen entwickelt, die in der Softwarefamilie berücksichtigt werden müssen. Das können neue finite Elemente, Solver linearer Gleichungssysteme, Präkonditionierer, Fehlerschätzer usw. sein. Anhand der in dieser Arbeit betrachteten numerischen Verfahren der Strömungsmechanik wurde das Ziel verfolgt, ein solches Modell zu entwickeln, dass die neuen Methoden ins Modell eingefügt werden können, ohne eine Modifizierung des Modells zu benötigen. Solche neuen Methoden können als neue Komponenten implementiert und in die Komponentenlisten (genauer Typlisten, Kap. 4) hinzugefügt werden. Aus solchen Listen wird eine neue Komponente in mehrere fertige Solver mit Hilfe von Generatoren beim Kompilieren automatisch hereingezogen, so dass nur ein minimaler Komponenten-Integrationsaufwand entsteht.

Die Betrachtung einer ganzen Softwarefamilie umfasst sofort sehr viele Verfahren und Kentnisse, die auch über mehrere Forschungsgebiete verbreitet sind. Um das Modell besser zu entwickeln sowie Abhängigkeiten und Parameter zu bestimmen, versucht man möglichst viele der bekannten Verfahren zu berücksichtigen, was gleichzei-

tig eine große Anzahl der zu implementierenden Komponenten bedeutet. Diese Tatsache kann von Kritikern der GP als ein Nachteil betrachtet werden [190]. Es ist aber in der GP sehr wichtig, die theoretische Modell-Entwicklung (Domain-Engineering) von der nachfolgenden Komponenten-Implementierung in einer Programmiersprache zu unterscheiden. Eine große Verfahrensvielfalt in der Modell-Entwicklung führt zu einem genaueren Domänen-Modell, bedeutet aber nicht, dass alle dieser Verfahren sofort implementiert werden müssen. Später implementierte Komponenten passen dann genau so wie die zuerst implementierte Komponenten zum gesamten geplannten Modell und sind genau so hoch wiederverwendbar. In diesem Zusammenhang enthält die erste Version der generativen Simulations-Solver (GSS) diejenigen Komponenten, die insbesondere gut parametrisierbar (durch Template-Parameter) sind und dadurch eine größere Verfahrens-Zahl mit einem kleinen Implementierungsaufwand abdecken, um die Mächtigkeit der statischen Konfiguration zu veranschaulichen.

- Die konformen finiten Elemente werden durch die Dimension, den Geometrie-Typ und die Ordnung der polynomialen Ansatzfunktionen bestimmt (Abschnitt 4.4), d.h. durch drei Parameter wird eine ganze Palette der FEM P_i und Q_i, $i = 0, 1, ...$ repräsentiert. Diese konformen Elemente werden in gemischter Form für die Diskretisierung der Navier-Stokes-Gleichungen benutzt, obwohl eine Anwendung in Einzelform für andere Gleichungen nicht ausgeschlossen ist. Nichtkonforme Elemente müssen dagegen einzeln definiert und die zugehörigen lokalen Matrizen generiert werden.
- Durch die Einteilung der Navier-Stokes-Gleichungen in die Einzelterme ist auch eine Lösung aller Sonderfälle möglich, z.B. Stokes-Gleichungen oder Euler-Gleichungen. Dabei fällt der zum fehlenden Gleichungsterm Codeteil (Behandlung der lokalen Matrix, etc.) bei der Kompilierung durch statische Konfiguration aus, was den entsprechenden Solver im Vergleich zum vollständigen NS-Solver wesentlich effizienter macht.
- Mit Hilfe der vorhandenen LASC-Bibliothek [118] wurden alle erwähnten iterativen Gleichungssystem-Solver (Tab. 5.4) ohne großen Aufwand implementiert. Im Laufen der Tests (Kap. 6) wurden allerdings die Residuum minimisierende Methoden bevorzugt, weil sie insbesondere mit dem gewählten Präkonditionierer (5.21) ein gutes Konvergenzverhalten zeigen. Andere, z.B. allgemeine ILU-Präkonditionierer, können evtl. mit anderen iterativen Verfahren besser zusammenpassen, was auch in der Kompilierungszeit konfiguriert wird.
- Für die erste GSS-Version wurde nur eine Gitter-Datenstruktur (R1, Abschnitt 5.2.1) implementiert und gebraucht. Andere Datenstrukturen können z.B. bei der Implementierung der h-Version (Gitteradaption) notwendig sein und müssen dann die R1-Datenstruktur in diesen Solver-Versionen ersetzen.
- Die Aufgaben-Dimension $d = 1, 2, 3$ genau wie der Geometrie-Typ (Dreiecke oder Vierecke, Abb. 5.8) sind Template-Parameter. Dashalb enthält z.B. ein 2D-Solver keine unnötige Codeteile der 3D-Version. Dasselbe gilt für den Geometrie-Typ und andere statische Daten.

Weitere allgemeine und Block-Präkonditionierer sowie nur theoretisch betrachtete

Konzepte: Partitionierung, Parallelisierung und Fehlerschätzung können später implementiert und in die Software integriert werden.

7.3 Symbolische Modellbildung

Der besondere Schwerpunkt dieser Arbeit liegt in der Definition von statischen Daten (Abschnitt 5.1), die schon vor der Kompilierung bekannt sein können. Die statischen Daten werden durch Template-Parameter implementiert, d.h. sie sind entweder ganze Zahlen oder durch Klassen-Templates repräsentiert, die selber nur weitere statische Daten enthalten dürfen. Im physischen Sinne stellen die statischen Daten mathematische Objekte dar, deren Behandlung nichts anderes als symbolische Berechnungen sind. Die guten Beispiele dafür sind Modellierung der diskreten Navier-Stokes-Gleichungen (Abschnitt 4.2) und der Aufbau polynomialer Ansatzfunktionen (Abschnitt 4.3).

Wir wollen nun den ganzen Weg solcher symbolischen Modellierung im entwickelten Simulations-Solver zusammenfassen. Beginnen wir mit den Navier-Stokes-Gleichungen, die zuerst in der Zeit diskretisiert, dann linearisiert und schließlich vereinfacht werden. Für jeden in den endgültigen diskreten Gleichungen enthaltenen Term wird eine lokale Matrix entsprechend dem gewählten Stokes-Element geladen (Abschnitt 5.2.3). Wenn das notwendige File nicht existiert, wird die Matrix mit Hilfe symbolisch gebildeter Ansatzfunktionen berechnet und für die Zukunft abgespeichert, so dass diese Berechnung nur einmal für das gewählte Stokes-Element stattfinden muss. Diese lokale Matrix wird bei der Assemblierung für jedes Gitter-Element in ein neues Koordinatensystem transformiert und zur globalen Steifigkeitsmatrix addiert (Abschnitt 5.2.4). Wenn diese globale Matrix mit allgemeinen iterativen Verfahren und Präkonditionierer gelöst werden könnte, wären keine weiteren symbolischen Definitionen bis zum Simulations-Ergebnis mehr nötig. Die Matrix muss aber in der Blockform (5.17) dargestellt werden, wobei zur Berechnung eines speziellen Block-Präkonditionierer (Abschnitt 5.2.6) die Matrix-Blöcke getrennt behandelt werden müssen. Da wir nicht nur inkompressible Navier-Stokes-Gleichungen betrachten, sondern auch Sonderfälle und je nach dem Sonderfall die Blöcke unterschiedliche Eigenschaften (z.B. symmetrisch oder nicht) haben können oder sogar ganz ausfallen (z.B. bei der Konvektions-Diffusions-Gleichung), werden die Matrix-Blöcke symbolisch bezeichnet und deren Zusammenhang mit den Gleichungstermen hergestellt. Es ist z.B. einfach zu merken, dass der Diffusions- und der Konvektions-Terme in den Matrix-Block F und der Druck-Term bzw. Kontinuität in die Blöcke B^T bzw. B beitragen. Die Form der diskreten Gleichungen (dort enthaltene Terme) bestimmt also die Gestalt der globalen Matrix schon in der Kompilierungszeit. Dadurch können ein passender Präkonditionierer bzw. notwendige iterative Verfahren automatisch gewählt werden. Für iterative Lösungen mit getrennten Matrix-Blöcken (z.B. zur Bestimmung von F^{-1}), die evtl. bei einer Block-Präkonditionierung notwendig sein können, sind die aus den Gleichungen stammenden Eigenschaften der Blöcke: Symmetrie und Definitheit entscheidend für die Wahl eines iterativen Verfahren. Einbeziehen einer weiteren Gleichung ins Modell, wie z.B. Energie-Gleichung, bedeutet neue Blöcke in der globalen Matrix wie auch eine andere präkonditionierte iterative Lösung, was analog statisch konfiguriert wird.

Die gekoppelte Matrix-Assemblierung und danach folgende Block-Zerlegung der

Matrix ist ziemlich allgemein, weil die Projektions-Methoden (Abschnitt 3.1.2), die Geschwindigkeit und Druck schon vor der Raumdiskretisierung entkoppeln, zu den bestimmten Block-Präkonditionierer äquivalent sind (Abschnitt 5.2.6, [130]). Diese Tatsache erleichtert wesentlich die Entwicklungsarbeit, weil man sich nur mit den verschiedenen Block-Präkonditionierern und nicht mit den neuen Gleichungen auseinander setzen muss.

7.4 Vor- und Nachteile der generativen Softwareentwicklung

Viele der Eigenschaften der GP wurden bereits in diesem Kapitel anhand des Beispiels der im Rahmen dieser Arbeit entwickelten Simulations-Solver erwähnt. Hier werden sie verallgemeinert und zusammengefasst:

+ Die Entwicklung eines Domänen-Modells umfasst viele Konzepte und Merkmale des Problembereichs, entdeckt deren Ähnlichkeiten und Unterschiede (in unserem Fall numerische Verfahren), parametrisiert diese und erstellt dadurch eine Grundlage für eine flexible Software mit vielen Optionen bzw. einer großen Verfahrensauswahl.
+ Möglichst viele der Modell-Parameter werden als statische Daten (Abschnitt 5.1) angenommen. Mit solchen Parameter wird die statische Konfiguration des Codes in der Kompilierungszeit durchgeführt, d.h. aus den vorhandenen wiederverwendbaren Komponenten werden nur notwendige Komponenten in eine oder mehrere fertige Softwaresysteme (Strömungssimulations-Solver) mit einbezogen. Dieses liefert eine hohe Effizienz von solchen Softwaresysteme unabhängig von der Optionen- bzw. Verfahrensvielfalt.
+ Die GP eröffnet einen besonderen Vorteil zur Behandlung mathematischer Objekte. Sie können symbolisch in der Kompilierungszeit behandelt werden, wie z.B. diskrete Navier-Stokes-Gleichungen (Abschnitt 4.2) oder polynomiale Ansatzfunktionen (Abschnitt 4.3). Dann werden solche mathematischen Objekte in der vereinfachten Form in den übersetzten Code hereingehen.
+ Formeln, die nur statische Daten benutzen, werden schon in der Kompilierungszeit berechnet, wie z.B. die ganzzahlige Potenzberechnung (Listing 4.1). Zu solchen kleinen Codeoptimierungs-Techniken auf dem unteren Implementierungs-Niveau gehört auch das Loop-Unrolling (Abschnitt 5.3.3, A.3).
− Zu den Nachteilen gehört die schnell mit der Anzahl der Optionen wachsende Kompilierungszeit, wenn insbesondere mehrere Softwaresystem-Varianten kompiliert werden. Deshalb muss man Berechnungen mit nichtlinearer Komplexität in der Kompilierungszeit vermeiden.
− Komplizierte Syntax der Template-Metaprogrammierung, die zur Umsetzung der statischen Konfiguration und der Kompilierungszeit-Berechnungen verwendet wird. Die Syntax basiert auf Klassen-Template-Definitionen und deren Spezialisierungen (Abschnitt 4.1), die ursprünglich zur keinen so extensiven Nutzung, wie in der Template-Metaprogrammierung, geplant wurden und mehr durch einen Zufall zu einer vollständigen Meta-Sprache geführt haben.

+ Mit Hilfe mehrerer kombinierbarer Komponenten und die dadurch entstehenden Softwaresystem-Varianten lassen sich Fehler im Code leichter suchen und lokalisieren. Beispielsweise werden verschiedene Stokes-Elemente mit verschiedenen Sonderfällen der Navier-Stokes-Gleichungen getestet, um zu erkennen, welcher der Gleichungsterme bzw. die dahinter stehende lokale Matrix zu einem Fehler führt.

7.5 Mögliche Weiterentwicklung

Die generative Softwareentwicklung gestattet nicht nur eine effiziente Umsetzung einer großen Zahl von Verfahren aus einem bestimmten Problembereich, sondern erstellt Beziehungen zu anderen Forschungsgebieten, wo der Simulations-Solver weiterentwickelt und eingesetzt werden kann.

Neue finite Elemente

Die Liste der implementierten und in der Arbeit erwähnten finiten Elemente kann man schnell erweitern, indem man z.B. die polynomiale Blasenfunktion (3.66) genau wie die konformen Ansatzfunktionen symbolisch definiert und eine Folge der stabilisierten konformen Elemente P_k^+/P_k und Q_k^+/Q_k, $k = 1, 2, ...$ erzeugt, wobei $+$ die Addition der Blasenfunktion bedeutet. Nichtkonforme und besondere Elemente müssen gesondert so definiert werden, dass für jeden Gleichungsterm eine lokale Matrix entsprechend diesem Element berechnet werden kann.

Weitere Gleichungssystemlöser und Präkonditionierer

Die Tests der konkreten 2D- und 3D-Simulationen (Kap. 6) haben gezeigt, dass der größte Rechenaufwand in der präkonditionierten iterativen Lösung, insbesondere derjenigen Gleichungssysteme, die bei der Simulationen instationärer Strömungen entstehen. Die richtigen Parameter und Kombinationen der iterativen Löser und Präkonditionierer können in vielen Fälle nur durch Tests bestimmt oder angepasst werden. Deshalb benötigt eine Weiterentwicklung in diese Richtung nicht nur theoretische Fortschritte, sondern auch viele Tests- und Benchmarks-Untersuchungen.

Parallelisierung

Trotz einer besonderen Sorgfalt bezüglich Recheneffizienz und Speicherverbrauch während der Softwareentwicklung erreicht man bei der Simulation reeller 3D-Strömungsmodelle sehr schnell die Rechenleistungs- und Speichergrenze. Beide Grenzen können durch die beschriebenen Parallelisierungs-Verfahren (Abschnitt 5.2.7) auf unterschiedlichen parallelen Architekturen [119] überschritten werden. Dadurch kann der GSS vor allem in eine neue Dimension der Ingenieur-Aufgaben einsteigen.

Neue Gleichungen

Zuerst muss man sich an die gemischten FEM mit Ansatzfunktionen gleicher Ordnung für die Geschwindigkeit und den Druck erinnern, die durch Einführung zusätzlicher Terme in der rechten Seite der Kontinuitätsgleichung stabilisiert werden. Solche

Terme müssen genau so wie äußere Kraft in der rechten Seite der Impulsgleichung als ein neuer Term symbolisch definiert werden. Für die neuen Terme werden dann die lokalen Vektoren generiert und am gesamten Assemblierungsprozess teilnehmen.

Eine andere Weiterentwicklung der Gleichungsmodelle besteht in der Umsetzung der im Abschnitt 4.2.3 beschriebenen Idee. Die als ganze Summanden im Modell der Navier-Stokes-Gleichungen dargestellten Terme (Diffusion, Konvektion, etc.) können weiter in Unbekannte, Konstante und Operatoren zerlegt werden, die dann auch symbolisch durch Klassen-Templates bezeichnet werden können. Daraus können neue Gleichungsterme bzw. neue Gleichungsmodelle gebildet werden. Die Beschreibung eines neuen Gleichungsmodells kann mit Macro-Definitionen in der C++ Sprache statt Template-Definitionen benutzerfreundlich vereinfacht werden. Dann wird ein effizienter FEM-Solver für eine als Macro beschriebene diskrete Gleichung, hinter welcher eine symbolische Template-Definition steht, automatisch nach der Kompilierung erzeugt. Solche Gleichungen können sowohl zur numerischen Strömungs-Simulation zusätzlich sein (z.B. Energie-Gleichung, verschiedene Turbulenz-Modelle) als auch aus anderen Forschungsgebieten stammen, die weit weg von der Strömungsmechanik liegen.

Adaptivität

Eine einfache Zeit-Adaption hat sich bei den numerischen Experimente (Kap. 6) mehrmals gut bewährt. Deshalb wäre eine Entwicklung anderer Adaption-Strategien sehr vorteilhaft. Als erstes denkt man an die h-Adaption, d.h. Gitter-Verfeinerung oder Vergröberung, was auch Implementierung guter Fehlerschätzer (Abschnitt 5.2.8) benötigt. Eine Kombination der Gitter-Adaption mit den finiten Elementen höherer Ordnung führt zur hp-Version der FEM, wobei die p-Version im GSS bereits vorhanden ist.

Eine andere und weniger bekannte Adaption-Strategie besteht in der Lösung unterschiedlicher Gleichungen auf verschiedenen Gitter-Partitionen [85]. In bestimmten Simulationsgebieten können also nicht nur die Navier-Stokes-Gleichungen, sondern deren Sonderfälle gelöst werden, wo die Strömung entsprechende Eigenschaften besitzt. Die Partitionierung muss hier nicht unbedingt im Sinne des Parallelisierungs-Konzepts verstanden werden, obwohl eine gleichzeitige Lösung unterschiedlicher Gleichungen auf verteilten Rechnern auch möglich ist, wobei das Load-Balancing-Problem (eine gleichgewichtete Verteilung des Gitters über Partitionen) noch komplizierter wird.

Datenbank der Simulations-Ergebnisse

Durch die gut kombinierbare Komponente steigt die Anzahl der möglichen Solver exponentiell, was schnell dazu führt, dass alle generierten Solver von einer Person nicht getestet werden können. Zu einem Kenntnis-Austausch ist es deshalb sinnvoll, eine Online-Datenbank zu entwickeln, wo die simulierte Modelle mit sämtlichen eingegebenen Solver-Parametern gespeichert werden können, um sowohl die Fehler im Solver zu entdecken als auch für die Lösung reeller Ingenieur-Probleme hilfreiche Information zu bekommen.

Literaturverzeichnis

[1] A. Agouzal, J. Baranger, J.-F. Maitre, and F. Oudin. Connection between finite volume and mixed finite element methods for a diffusion problem with nonconstant coefficients. Application to a convection diffusion problem. *East-West J. Numer. Math.*, 3(4):237–254, 1995.

[2] J. Alberty, C. Carstensen, and S. A. Funken. Remarks around 50 lines of Matlab: short finite element implementation. *Numer. Algorithms*, 20(2-3):117–137, 1999.

[3] A. Alexandrescu. *Modern C++ Design*. Addison-Wesley Publishing Co., Boston, 2001.

[4] A. Alexandrescu. Typelists and Applications. *C/C++ Users Journal*, 2, 2002.

[5] T. Apel and N. Düvelmeyer. Transformation of hexahedral finite element meshes into tetrahedral meshes according to quality criteria. Technical Report 03-09, Technische Universität Chemnitz, SFB 393, May 2003.

[6] T. Apel, G. Haase, A. Meyer, and M. Pester. Parallel solution of finite element equation system: efficient inter-processor communication. Technical Report SPC 95-5, Technische Universität Chemnitz, February 1995.

[7] T. Apel, S. Nicaise, and J. Schöberl. Crouzeix-Raviart type elements on anisotropic meshes. Technical Report 99-10, Technische Universität Chemnitz, SFB 393, Mai 1999.

[8] F. Aurenhammer and R. Klein. Voronoi diagrams. In J. Sack and G. Urrutia, editors, *Handbook of Computational Geometry*, pages 201–290. Elsevier Science Publishing, 2000. [SFB Report F003-092, TU Graz, Austria, 1996].

[9] I. Babuška and W. C. Rheinboldt. Error estimates for adaptive finite element computations. *SIAM J. Numer. Anal.*, 15(4):736–754, 1978.

[10] G. Bader and K.-D. Krannich. Einführung in das parallele Rechnen. Scriptum zur Vorlesung, Brandenburgische Technische Universität, Cottbus, 1997.

[11] R. E. Bank and R. K. Smith. A posteriori error estimates based on hierarchical bases. *SIAM J. Numer. Anal.*, 30(4):921–935, 1993.

[12] J. Baranger, J.-F. Maitre, and F. Oudin. Connection between finite volume and mixed finite element methods. *RAIRO Modél. Math. Anal. Numér.*, 30(4):445–465, 1996.

[13] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods.* SIAM, Philadelphia, 1993.

[14] J. J. Barton and L. R. Nackman. *Scientific and engineering C++: an introduction with advanced techniques and examples.* Addison-Wesley, Reading, MA, USA, 1994.

[15] R. Becker. An optimal-control approach to a posteriori error estimation for finite element discretizations of the Navier-Stokes equations. *East-West J. Numer. Math.*, 8(4):257–274, 2000.

[16] R. Becker, M. Braack, and R. Rannacher. Adaptive finite element methods for flow problems. In *Foundations of computational mathematics (Oxford, 1999)*, volume 284 of *London Math. Soc. Lecture Note Ser.*, pages 21–44. Cambridge Univ. Press, Cambridge, 2001.

[17] R. Becker and R. Rannacher. An optimal control approach to a posteriori error estimation in finite element methods. *Acta Numer.*, 10:1–102, 2001.

[18] G. Berti. Generic components for grid data structures and algorithms with C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[19] G. Berti. *Generic software components for scientific computing.* PhD thesis, Cottbus: Brandenburgische TU Cottbus, Fakultät Mathematik, Naturwissenschaften u. Informatik, 2000.

[20] BLAS: Basic Linear Algebra Subprograms.
`http://www.netlib.org/blas/`.
Stand: 16.02.2005.

[21] D. Boffi. Stability of higher order triangular Hood-Taylor methods for the stationary Stokes equations. *Math. Models Methods Appl. Sci.*, 4(2):223–235, 1994.

[22] D. Boffi. Three-dimensional finite element methods for the Stokes problem. *SIAM J. Numer. Anal.*, 34(2):664–670, 1997.

[23] M. Bollhöfer. A robust ILU based on monitoring the growth of the inverse factors. Technical Report 00-31, Technische Universität Chemnitz, SFB 393, July 2000.

[24] L. A. Bordag, O. G. Chkhetiani, M. Fröhner, and V. Myrnyy. Interaction of a rotational motion and an axial flow in small geometries for a couette-taylor problem. *Journal of Fluids and Structures*, 20(5):621–641, 2005.

[25] F. Brezzi and R. S. Falk. Stability of higher-order Hood-Taylor methods. *SIAM J. Numer. Anal.*, 28(3):581–590, 1991.

[26] F. Brezzi and M. Fortin. *Mixed and hybrid finite element methods*, volume 15 of *Springer Series in Computational Mathematics*. Springer-Verlag, New York, 1991.

[27] G. F. Carey. *Computational grids*. Series in Computational and Physical Processes in Mechanics and Thermal Sciences. Taylor & Francis, Washington, DC, 1997.

[28] C. Carstensen. All first-order averaging techniques for a posteriori finite element error control on unstructured grids are efficient and reliable. *Math. Comp.*, 73(247):1153–1165 (electronic), 2004.

[29] C. Carstensen. Some remarks on the history and future of averaging techniques in a posteriori finite element error analysis. *ZAMM Z. Angew. Math. Mech.*, 84(1):3–21, 2004.

[30] T. F. Chan, E. Gallopoulos, V. Simoncini, T. Szeto, and C. H. Tong. A quasi-minimal residual variant of the Bi-CGSTAB algorithm for nonsymmetric systems. *SIAM J. Sci. Comput.*, 15(2):338–347, 1994.

[31] P. Chin and P. Forsyth. A comparison of GMRES and CGSTAB accelerations for incompressible Navier- Stokes problems. *J. Comput. Appl. Math.*, 46(3):415–426, 1993.

[32] S. H. Chou and D. Y. Kwak. Analysis and convergence of a MAC-like scheme for the generalized Stokes problem. *Numer. Methods Partial Differential Equations*, 13(2):147–162, 1997.

[33] S. H. Chou and D. Y. Kwak. A covolume method based on rotated bilinears for the generalized Stokes problem. *SIAM J. Numer. Anal.*, 35(2):494–507 (electronic), 1998.

[34] S.-H. Chou, D. Y. Kwak, and P. S. Vassilevski. Mixed covolume methods for elliptic problems on triangular grids. *SIAM J. Numer. Anal.*, 35(5):1850–1861 (electronic), 1998.

[35] A. L. Coutinho, M. A. Martins, J. L. Alves, L. Landau, and A. Moraes. Edge-based finite element techniques for nonlinear solid mechanics problems. *Int. J. Numer. Methods Eng.*, 50(9):2053–2068, 2001.

[36] E. Creuse, G. Kunert, and S. Nicaise. A posteriori error estimation for the Stokes Problem: anisotropic and isotropic diskretization. Technical Report 03-01, Technische Universität Chemnitz, SFB 393, January 2003.

[37] K. Czarnecki. Generative programming and software system families. In W. Taha, editor, *Semantics, applications, and implementation of program generation. 2nd international workshop, SAIG 2001, Florence, Italy*, Lect. Notes Comput. Sci. 2196. Berlin: Springer, 2001.

[38] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools and applications.* Addison-Wesley, 2000.

[39] K. Czarnecki, K. Osterbye, and M. Völter. Generative programming. In J. e. a. Hernndez, editor, *Object-oriented technology. ECOOP 2002 workshop reader. ECOOP 2002 workshops and posters, Mlaga, Spain*, Lect. Notes Comput. Sci. 2548, pages 15–29. Berlin: Springer, 2002.

[40] H. Daniels and A. Peters. PASTIS-3D – A parallel finite element projection code for the time-dependent incompressible Navier-Stokes equations. In *Hebeker, Friedrich-Karl (ed.) et al., Numerical methods for the Navier-Stokes equations. Proceedings of the international workshop, Heidelberg, Germany, October 25-28, 1993.*, number 47 in Notes on Numerical Fluid Mechanics, pages 31–39. Vieweg, Braunschweig, 1994.

[41] E. J. Dean and R. Glowinski. On some finite element methods for the numerical simulation of incompressible viscous flow. In M. D. Gunzburger and R. A. Nicolaides, editors, *Incompressible computational fluid dynamics: trends and advances*, pages 17–65. Cambridge University Press, 1993.

[42] R. Deiterding. *Parallel adaptive simulation of multi-dimensional detonation structures*. PhD thesis, Cottbus: Brandenburgische TU Cottbus, Fakultät Mathematik, Naturwissenschaften u. Informatik, 2003.

[43] U. W. Eisenecker. Generative programming with C++. In H. Mössenböck, editor, *Modular Programming Languages (JMLC97, Linz, Austria, March 1997)*, pages 351–365. Springer-Verlag, 1997.

[44] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison–Wesley, Reading, MA, 1990.

[45] H. Elman and D. Silvester. Fast nonsymmetric iterations and preconditioning for Navier-Stokes equations. *SIAM J. Sci. Comput.*, 17(1):33–46, 1996.

[46] H. C. Elman. Preconditioning for the steady-state Navier–Stokes equations with low viscosity. *SIAM J. Sci. Comput.*, 20(4):1299–1316, 1999.

[47] H. C. Elman, D. Loghin, and A. J. Wathen. Preconditioning techniques for Newton's method for the incompressible Navier-Stokes equations. *BIT*, 43(suppl.):961–974, 2003.

[48] H. C. Elman, D. J. Silvester, and A. J. Wathen. Block preconditioners for the discrete incompressible Navier-Stokes equations. *Internat. J. Numer. Methods Fluids*, 40(3-4):333–344, 2002. ICFD Conference on Numerical Methods for Fluid Dynamics, Part II (Oxford, 2001).

[49] H. C. Elman, D. J. Silvester, and A. J. Wathen. Performance and analysis of saddle point preconditioners for the discrete steady-state Navier-Stokes equations. *Numer. Math.*, 90(4):665–688, 2002.

[50] L. Fatone, P. Gervasio, and A. Quarteroni. Multimodels for incompressible flows. *J. Math. Fluid Mech.*, 2(2):126–150, 2000.

[51] L. Fatone, P. Gervasio, and A. Quarteroni. Multimodels for incompressible flows: iterative solutions for the Navier-Stokes/Oseen coupling. *M2AN Math. Model. Numer. Anal.*, 35(3):549–574, 2001.

[52] C. A. Felippa. Introduction to Finite Element Methods. Technical report, University of Colorado at Boulder, Department of Aerospace Engineering Sciences, 2003.

[53] J. H. Ferziger and M. Perić. *Computational methods for fluid dynamics.* Springer-Verlag, Berlin, revised edition, 1999.

[54] L. P. Franca, T. J. Hughes, and R. Stenberg. Stabilized finite element methods. In M. D. Gunzburger and R. A. Nicolaides, editors, *Incompressible computational fluid dynamics: trends and advances*, pages 87–107. Cambridge University Press, 1993.

[55] R. W. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM J. Sci. Comput.*, 14(2):470–482, 1993.

[56] R. W. Freund and N. M. Nachtigal. QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Numer. Math.*, 60(3):315–339, 1991.

[57] O. Friedrich. A new method for generating inner points of triangulations in two dimensions. *Comput. Methods Appl. Mech. Eng.*, 104(1):77–86, 1993.

[58] M. Fröhner. Numerische Methoden in der Hydrodynamik. *Wissenschaftliche Schriftenreihe der Technischen Hochschule Karl-Marx-Stadt*, 12:68, 1984.

[59] M. Fröhner. Numerical solution of the Burgers equation by a semidiscrete spline- Galerkin technique. In *BAIL IV, Proc. 4th Int. Conf. Boundary and interior layers, Novosibirsk/USSR 1986, Conf. Ser. 10*, pages 292–298. 1986.

[60] M. Fröhner. Numerical methods for diffusion-convection equations. In *Differential equations and optimal control, Proc. 7th Reg. Sci. Sess., Kalsk/Pol. 1988, Sect.: Differential equations*, pages 7–14. 1989.

[61] GAMBIT CFD Preprocessor.
`http://www.fluent.com/software/gambit/`.
Stand: 1.08.2005.

[62] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns. Elements of reusable object-oriented software.* Addison-Wesley professional computing series. Addison-Wesley, Reading, MA, 1995.

[63] R. V. Garimella. Mesh data structure selection for mesh generation and FEA applications. *Int. J. Numer. Methods Eng.*, 55(4):451–478, 2002.

[64] A. Gauthier, F. Saleri, and A. Veneziani. A fast preconditioner for the incompressible Navier-Stokes equations. *Comput. Vis. Sci.*, 6(2-3):105–112, 2004.

[65] P. L. George. Automatic mesh generation and finite element computation. In *Handbook of numerical analysis, Vol. IV*, Handb. Numer. Anal., IV, pages 69–190. North-Holland, Amsterdam, 1996.

[66] P.-L. George and H. Borouchaki. *Delaunay triangulation and meshing.* Editions Hermès, Paris, 1998.

[67] K. Gersten. *Einführung in die Strömungsmechanik. Mit 96 Bildern, 10 Tabellen und 52 durchgerechneten Beispielen. 6. überarb. Aufl.* Friedr. Vieweg & Sohn, Braunschweig, 1991.

[68] V. Girault and P.-A. Raviart. *Finite element methods for Navier-Stokes equations. Theory and algorithms*, volume 5 of *Springer Series in Computational Mathematics*. Springer, Berlin, 1986.

[69] A. Greenbaum. *Iterative methods for solving linear systems*, volume 17 of *Frontiers in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.

[70] P. Gresho and R. Sani. *Incompressible flow and the finite element method. Vol. 1: Advection-diffusion. Vol. 2: Isothermal laminar flow.* Chichester: Wiley, 2000.

[71] P. M. Gresho. On the theory of semi-implicit projection methods for viscous incompressible flow and its implementation via a finite element method that also introduces a nearly consistent mass matrix. I. Theory. *Internat. J. Numer. Methods Fluids*, 11(5):587–620, 1990. Computational methods in flow analysis (Okayama, 1988).

[72] P. M. Gresho and S. T. Chan. On the theory of semi-implicit projection methods for viscous incompressible flow and its implementation via a finite element method that also introduces a nearly consistent mass matrix. II. Implementation. *Internat. J. Numer. Methods Fluids*, 11(5):621–659, 1990. Computational methods in flow analysis (Okayama, 1988).

[73] M. Griebel, T. Dornseifer, and T. Neunhoeffer. *Numerische Simulation in der Strömungsmechanik. Eine praxisorientierte Einführung.* Vieweg, Wiesbaden, 1995.

[74] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface.* Scientific and Engineering Computation. MIT Press, Cambridge, MA, USA, Nov. 1999.

[75] M. D. Gunzburger. *Finite element methods for viscous incompressible flows.* Computer Science and Scientific Computing. Academic Press, Boston, MA, 1989.

[76] G. Haase. *Parallelisierung numerischer Algorithmen für partielle Differentialgleichungen.* B. G. Teubner, Stuttgart, 1999.

[77] W. Habashi, M. Peeters, M. Robichaud, and V.-N. Nguyen. A fully-coupled finite element algorithm, using direct and iterative solvers, for the incompressible navier-stokes equations. In M. D. Gunzburger and R. A. Nicolaides, editors, *Incompressible computational fluid dynamics: trends and advances*, pages 151–181. Cambridge University Press, 1993.

[78] W. Hackbusch. On first and second order box schemes. *Computing*, 41(4):277–296, 1989.

[79] L. Hemmingsson-Frändén and A. Wathen. A nearly optimal preconditioner for the Navier-Stokes equations. *Numer. Linear Algebra Appl.*, 8(4):229–243, 2001.

[80] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Research Nat. Bur. Standards*, 49:409–436 (1953), 1952.

[81] V. Heuveline and R. Rannacher. Duality-based adaptivity in the *hp*-finite element method. *J. Numer. Math.*, 11(2):95–113, 2003.

[82] J. G. Heywood and R. Rannacher. Finite element approximation of the nonstationary Navier-Stokes problem. I. Regularity of solutions and second-order error estimates for spatial discretization. *SIAM J. Numer. Anal.*, 19(2):275–311, 1982.

[83] R. Hirschfeld. *Ein Beitrag zur Modellierung objektorientierter Softwarearchitekturen durch Einfürung einer Komponentenverknüpfungsschicht*. PhD thesis, TU Ilmenau, 1997.

[84] S. Idelsohn and E. Onate. Finite volumes and finite elements: Two 'good friends'. *Int. J. Numer. Methods Eng.*, 37(19):3323–3341, 1994.

[85] J. G. Kallinderis and J. R. Baron. Adaptation methods for viscous flows. In *Computational methods in viscous aerodynamics*, pages 163–196. Elsevier, Amsterdam, 1990.

[86] J. G. Kallinderis and J. R. Baron. The finite volume approach for the Navier-Stokes equations. In *Computational methods in viscous aerodynamics*, pages 117–146. Elsevier, Amsterdam, 1990.

[87] Y. Kallinderis. A 3-D finite-volume method for the Navier-Stokes equations with adaptive hybrid grids. *Appl. Numer. Math.*, 20(4):387–406, 1996. Adaptive mesh refinement methods for CFD applications (Atlanta, GA, 1994).

[88] G. Karypis. Metis: Family of multilevel partitioning algorithms.
`http://www-users.cs.umn.edu/~karypis/metis/`.
Stand: 28.01.2005.

[89] D. Kay, D. Loghin, and A. Wathen. A preconditioner for the steady-state Navier-Stokes equations. *SIAM J. Sci. Comput.*, 24(1):237–256 (electronic), 2002.

[90] C. T. Kelley. *Iterative methods for linear and nonlinear equations*, volume 16 of *Frontiers in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1995.

[91] P. Knabner and L. Angermann. *Numerik partieller Differentialgleichungen. Eine anwendungsorientierte Einführung.* Springer, Berlin, 2000.

[92] P. Knupp and S. Steinberg. *Fundamentals of grid generation*. CRC Press, Boca Raton, FL, 1994.

[93] U. Küster. Hardware environments for high performance computing. *Proceedings of CIT-2004, Comp. Tech., Novosibirsk*, 9:50–58, 2004.

[94] O. A. Ladyzhenskaya. *The mathematical theory of viscous incompressible flow.* Second English edition, revised and enlarged. Translated from the Russian by Richard A. Silverman and John Chu. Mathematics and its Applications, Vol. 2. Gordon and Breach Science Publishers, New York, 1969.

[95] L. Landau and E. Lifschitz. *Lehrbuch der theoretischen Physik. Band VI: Hydrodynamik. In deutscher Sprache hrsg. von Wolfgang Weller. Übers. aus dem Russ. von Adolf Kühnel und Wolfgang Weller. 5. überarb. Aufl.* H. Deutsch, Frankfurt am Main, 1991.

[96] A. Langer and K. Kreft. C++ Expression Templates. *C/C++ Users Journal*, 3, 2003.

[97] C. Li and K. Vuik. Eigenvalue analysis of the SIMPLE preconditioning for incompressible flow. Report 02-15, Delft University of Technology, Department of Applied Mathematical Analysis, Delft, 2002.

[98] Z.-C. Li and S. Wang. The finite volume method and application in combinations. *J. Comput. Appl. Math.*, 106(1):21–53, 1999.

[99] V. D. Liseikin. *Grid generation methods*. Scientific Computation. Springer-Verlag, Berlin, 1999.

[100] The ASCI LKF Benchmark Code.
`http://www.llnl.gov/asci_benchmarks/asci/limited/lfk/`.
Stand: 16.02.2005.

[101] D. Loghin and A. Wathen. Preconditioning the advection-diffusion equation: the Green's function approach. Technical Report NA-97/15, Oxford University Computing Laboratory, 1997.

[102] D. Loghin and A. J. Wathen. Schur complement preconditioners for the Navier-Stokes equations. *Internat. J. Numer. Methods Fluids*, 40(3-4):403–412, 2002. ICFD Conference on Numerical Methods for Fluid Dynamics, Part II (Oxford, 2001).

[103] R. Löhner. Design of incompressible flow solvers: practical aspects. In M. D. Gunzburger and R. A. Nicolaides, editors, *Incompressible computational fluid dynamics: trends and advances*, pages 267–293. Cambridge University Press, 1993.

[104] G. I. Marchuk. Splitting and alternating direction methods. In *Handbook of numerical analysis, Vol. I*, Handb. Numer. Anal., I, pages 197–462. North-Holland, Amsterdam, 1990.

[105] K.-A. Mardal and R. Winther. Uniform preconditioners for the time dependent Stokes problem. *Numer. Math.*, 98(2):305–327, 2004.

[106] M. A. Martins, J. L. Alves, and A. L. Coutinho. Parallel edge-based finite element techniques for nonlinear solid mechanics. In Palma, Jose M. L. M. et al., editor, *Vector and parallel processing - VECPAR 2000. 4th international conference, Porto, Portugal, June 21-23, 2000. Selected papers and invited talks*, Lect. Notes Comput. Sci. 1981, pages 506–518. Springer, Berlin, 2001.

[107] M. A. D. Martins, A. L. G. A. Coutinho, and J. L. D. Alves. Parallel iterative solution of finite element systems of equations employing edge-based data structures. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing (Minneapolis, MN, 1997)*, page 8 (electronic), Philadelphia, PA, 1997. SIAM.

[108] G. Medić and B. Mohammadi. NSIKE - an incompressible Navier-Stokes solver for unstructured meshes. Technical Report RR-3644, INRIA, March 1999.

[109] S. Meinel. Untersuchungen zur Druckiterationsverfahren für dichteveränderliche Strömungen mit niedriger Machzahl. Technical Report 00-16, Technische Universität Chemnitz, SFB 393, März 2000.

[110] A. Meister. Comparison of different Krylov subspace methods embedded in an implicit finite volume scheme for the computation of viscous and inviscid flow fields on unstructured grids. *J. Comput. Phys.*, 140(2):311–345, 1998.

[111] A. Meister. *Numerik linearer Gleichungssysteme. Eine Einführung in moderne Verfahren.* Friedr. Vieweg & Sohn, Braunschweig, 1999.

[112] A. Meyer. A parallel preconditioned conjugate gradient method using domain decomposition and inexact solvers on each subdomain. *Computing*, 45(3):217–234, 1990.

[113] A. Meyer and M. Pester. Verarbeitung von Sparse-Matrizen in Kompaktspeicherform. Technical Report SPC 94-12, Technische Universität Chemnitz, Juni 1994.

[114] Intel Math Kernel Library.
`http://developer.intel.com/software/products/mkl/`.
Stand: 16.02.2005.

[115] *MPI: A Message-Passing Interface Standard*, June 12, 1995.

[116] *MPI-2: Extensions to the Message-Passing Interface*, July 18, 1997.

[117] M. F. Murphy, G. H. Golub, and A. J. Wathen. A note on preconditioning for indefinite linear systems. *SIAM J. Sci. Comput.*, 21(6):1969–1972, 2000.

[118] V. Myrnyy. LASC: Linear Algebra for Scientific Computing.
`http://www.scientificcpp.com/lasc/`.
Stand: 16.02.2005.

[119] V. Myrnyy. Parallelisierung numerischer Probleme - Möglichkeiten und Grenzen. Technical Report M-11/2001, Brandenburgische TU Cottbus, Fakultät Mathematik, Naturwissenschaften und Informatik, 29 p. , November 2001.

[120] V. Myrnyy. Simulation of newtonian molecular dynamic of gas with the help of parallel computing (russian). *The Bulletin of KazSU, ser. math., mech., inf., Almaty, Kazakhstan*, 2(25):112–119, 2001.

[121] V. Myrnyy. Recursive function templates as a solution of linear algebra expressions in C++. *arXiv.org e-Print*, cs.MS/0302026, 2003.

[122] V. Myrnyy. C++ metaprogramming in scientific computing. *Proceedings of CIT-2004, Comp. Tech., Novosibirsk*, 9:70–78, 2004.

[123] V. Myrnyy. Typelists & C++ Polynomial Meta-Arithmetic. *C/C++ Users Journal*, 8:22–29, 2004.

[124] V. Myrnyy and M. Fröhner. On a program of molecular dynamic simulation of gas with the elements of parallel algorithms (russian). *Comp. Tech., Novosibirsk*, 6(3):32–50, 2001.

[125] T. Neubert. Anwendung von generativen Programmiertechniken am Beispiel der Matrixalgebra. Master's thesis, TU Chemnitz, April, 1998.

[126] The Open Source Software Project based on IBM's Visualization Data Explorer.
`http://www.opendx.org`.
Stand: 1.08.2005.

[127] C. C. Paige and M. A. Saunders. Solutions of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12(4):617–629, 1975.

[128] F. Pascal and J.-M. Ghidaglia. Footbridge between finite volumes and finite elements with applications to CFD. *Internat. J. Numer. Methods Fluids*, 37(8):951–986, 2001.

[129] S. V. Patankar. *Numerical heat transfer and fluid flow.* Series in Computational Methods in Mechanics and Thermal Sciences. McGraw-Hill, New York, 1980.

[130] J. B. Perot. An analysis of the fractional step method. *J. Comput. Phys.*, 108(1):51–58, 1993.

[131] M. Pester. Bibliotheken zur Entwicklung paralleler Algorithmen. Technical Report 02-01, Technische Universität Chemnitz, SFB 393, Januar 2002.

[132] R. C. Peterson. *The numerical solution of free-surface problems for incompressible newtonian fluids.* PhD thesis, University of Leeds, School of Computer Studies, 1999.

[133] L. Quartapelle. *Numerical solution of the incompressible Navier-Stokes equations.* International Series of Numerical Mathematics. Birkhäuser, Basel, 1993.

[134] A. Quarteroni, F. Saleri, and A. Veneziani. Approximation of Navier-Stokes equations via algebraic factorizations. In *Navier-Stokes equations: theory and numerical methods (Varenna, 1997)*, volume 388 of *Pitman Res. Notes Math. Ser.*, pages 322–334. Longman, Harlow, 1998.

[135] A. Quarteroni, F. Saleri, and A. Veneziani. Analysis of the Yosida method for the incompressible Navier-Stokes equations. *J. Math. Pures Appl. (9)*, 78(5):473–503, 1999.

[136] A. Quarteroni, F. Saleri, and A. Veneziani. Factorization methods for the numerical approximation of Navier-Stokes equations. *Comput. Methods Appl. Mech. Engrg.*, 188(1-3):505–526, 2000.

[137] A. Quarteroni and A. Valli. *Numerical approximation of partial differential equations.* Springer Series in Computational Mathematics. Springer-Verlag, Berlin, 1994.

[138] A. Quarteroni and A. Valli. *Domain decomposition methods for partial differential equations.* Numerical Mathematics and Scientific Computation. Oxford: Clarendon Press. xv, 360 p., 1999.

[139] A. Ramage. A multigrid preconditioner for stabilised discretisations of advection-diffusion problems. *J. Comput. Appl. Math.*, 110(1):187–203, 1999.

[140] R. Rannacher. Finite Element Methods for the Incompressible Navier-Stokes Equations. Technical Report 99-37 (SFB 359), Universität Heidelberg, September 1999.

[141] R. Rannacher. Adaptive finite element methods for partial differential equations. In *Proceedings of the International Congress of Mathematicians, Vol. III (Beijing, 2002)*, pages 717–726, Beijing, 2002. Higher Ed. Press.

[142] R. Rannacher and S. Turek. Simple nonconforming quadrilateral Stokes element. *Numer. Methods Partial Differential Equations*, 8(2):97–111, 1992.

[143] Y. Saad. Preconditioning techniques for nonsymmetric and indefinite linear systems. *J. Comput. Appl. Math.*, 24(1-2):89–105, 1988. Iterative methods for the solution of linear systems.

[144] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, 1993.

[145] Y. Saad. *Iterative methods for sparse linear systems.* PWS Publishing Company, Boston, 1995.

[146] Y. Saad, O. Axelsson, I. Duff, W.-P. Tang, H. van der Vorst, and A. Wathen. Preconditioning techniques for large sparse matrix problems in industrial applications, SPARSE '99. *Numer. Linear Algebra Appl.*, 7(7-8):489–490, 2000.

[147] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7(3):856–869, 1986.

[148] Y. Saad and J. Zhang. Enhanced multi-level block ILU preconditioning strategies for general sparse linear systems. *J. Comput. Appl. Math.*, 130(1-2):99–118, 2001.

[149] M. Schäfer and S. Turek. Benchmark computations of laminar flow around a cylinder. Technical Report 96-03, Universität Heidelberg, SFB 359, 1996.

[150] H. Schlichting and K. Gersten. *Grenzschicht-Theorie. 9., völlig neu bearb. u. erw. Aufl.* Springer, Berlin, 1997.

[151] P. Schreiber. *Eine nichtkonforme Finite-Elemente-Methode zur Lösung der inkompressiblen 3-D Navier-Stokes Gleichungen.* PhD thesis, Univ. Heidelberg, Naturwiss.-Math. Fak., Heidelberg, 1996.

[152] H. R. Schwarz. *Methode der finiten Elemente. Eine Einführung unter besonderer Berücksichtigung der Rechenpraxis. 3., neubearb. Aufl.* Leitfäden der Angewandten Mathematik und Mechanik, 47. Teubner Studienbücher Mathematik, Stuttgart, 1991.

[153] D. Silvester, H. Elman, D. Kay, and A. Wathen. Efficient preconditioning of the linearized Navier-Stokes equations for incompressible flow. *J. Comput. Appl. Math.*, 128(1-2):261–279, 2001. Numerical analysis 2000, Vol. VII, Partial differential equations.

[154] D. Silvester and A. Wathen. Fast iterative solution of stabilised Stokes systems. II. Using general block preconditioners. *SIAM J. Numer. Anal.*, 31(5):1352–1367, 1994.

[155] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 10(1):36–52, 1989.

[156] B. Stroustrup. *The C++ Programming Language: Third Edition.* Addison-Wesley Publishing Co., Reading, Mass., 1997.

[157] R. M. Sydenstricker, M. A. Martins, A. L. Coutinho, and J. L. Alves. Edge-based interface elements for solution of three-dimensional geomechanical problems. In Palma, Jose M. L. M. et al., editor, *High performance computing for computational science - VECPAR 2002. 5th international conference, Porto, Portugal, June 26-28, 2002. Selected papers and invited talks*, Lect. Notes Comput. Sci. 2565, pages 53–64. Springer, Berlin, 2003.

[158] C. Taylor and P. Hood. A numerical solution of the Navier-Stokes equations using the finite element technique. *Internat. J. Comput. & Fluids*, 1(1):73–100, 1973.

[159] C. Taylor, J. Rance, and J. Midwell. A note on the imposition of traction boundary conditions when using the FEM for the incompressible flow problems. *Comm. Applied Num. Meth.*, 1(1):113–121, 1985.

[160] R. Temam. *Navier-Stokes equations. Theory and numerical analysis. 3rd (rev.) ed.* Studies in Mathematics and its Applications, Vol. 2. North-Holland, Amsterdam, 1984.

[161] R. Thatcher. The finite element method for three dimensional incompressible flow. In M. D. Gunzburger and R. A. Nicolaides, editors, *Incompressible computational fluid dynamics: trends and advances*, pages 427–445. Cambridge University Press, 1993.

[162] J. F. Thompson, B. K. Soni, and N. P. Weatherill, editors. *Handbook of grid generation.* CRC Press, Boca Raton, FL, 1999.

[163] S. Turek. *Efficient solvers for incompressible flow problems. An algorithmic and computational approach.* Lecture Notes in Computational Science and Engineering. Springer, Berlin, 1999.

[164] OMG Unified Modelling Language Specification. `http://www.uml.org`, March 2003. Version 1.5.

[165] E. Unruh. Prime number computation. *ANSI X3J16-94-0075/ISO WG21-462*, 1994.

[166] H. A. van der Vorst. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 13(2):631–644, 1992.

[167] R. Vanselow. Relations between FEM and FVM applied to the Poisson equation. *Computing*, 57(2):93–104, 1996.

[168] R. Vanselow and H.-P. Scheffler. Convergence analysis of a finite volume method via a new nonconforming finite element method. *Numer. Methods Partial Differential Equations*, 14(2):213–231, 1998.

[169] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[170] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[171] T. Veldhuizen. Techniques for scientific C++. Technical Report #542, Version 0.4, Indiana University, August 2000.

[172] T. Veldhuizen and K. Ponnambalam. Linear algebra with C++ template metaprograms. *Dr. Dobb's Journal of Software Tools*, 21(8):38–44, Aug. 1996.

[173] A. Veneziani. Block factorized preconditioners for high-order accurate in time approximation of the Navier-Stokes equations. *Numer. Methods Partial Differential Equations*, 19(4):487–510, 2003.

[174] R. Verfürth. A posteriori error estimators and adaptive mesh-refinement techniques for the navier-stokes equations. In M. D. Gunzburger and R. A. Nicolaides, editors, *Incompressible computational fluid dynamics: trends and advances*, pages 447–475. Cambridge University Press, 1993.

[175] R. Verfürth. The equivalence of a posteriori error estimators. In *Fast solvers for flow problems (Kiel, 1994)*, volume 49 of *Notes Numer. Fluid Mech.*, pages 273–283. Vieweg, Braunschweig, 1995.

[176] R. Verfürth. *A review of a posteriori error estimation and adaptive mesh-refinement techniques.* Wiley-Teubner Series Advances in Numerical Mathematics. John Wiley & Sons., 1996.

[177] R. Verfürth. Numerische strömungsmechanik. Vorlesungsskriptum, Ruhr-Universität Bochum, 1998.

[178] P. Vijayan and Y. Kallinderis. A 3D finite-volume scheme for the Euler equations on adaptive tetrahedral grids. *J. Comput. Phys.*, 113(2):249–267, 1994.

[179] C. Vuik and A. Saghir. The Krylov accelerated SIMPLE(R) method for incompressible flow. Report 02-01, Delft University of Technology, Department of Applied Mathematical Analysis, Delft, 2002.

[180] C. Vuik, A. Saghir, and G. Boerstoel. The Krylov accelerated SIMPLE(R) method for flow problems in industrial furnaces. *Int. J. for Num. Meth. Fluids*, 33:1027–1040, 2000.

[181] C. Vuik and G. Segal. A simple iterative linear solver for the 3D incompressible Navier-Stokes equations discretized by the finite element method. Report 95-64, Delft University of Technology, Department of Applied Mathematical Analysis, Delft, 1995.

[182] C. Walshaw. Jostle - graph partitioning software.
`http://staffweb.cms.gre.ac.uk/~c.walshaw/jostle/`.
Stand: 28.01.2005.

[183] A. Wathen. Preconditioning and fast solvers for incompressible flow. Technical Report NA-04/08, Oxford University Computing Laboratory, 2004.

[184] A. Wathen and D. Silvester. Fast iterative solution of stabilised Stokes systems. I. Using simple diagonal preconditioners. *SIAM J. Numer. Anal.*, 30(3):630–649, 1993.

[185] W. Weinelt, H. Kretzschmar, F. Kuhnert, M. Fröhner, and B. Heinrich. Some results of the finite difference method for hyperbolic and elliptic problems. In *Mathematical structures, computational mathematics, mathematical modelling 2*, pages 19–28. Pap. dedic. L. Iliev 70th Anniv., 1984.

[186] Wikipedia, the free encyclopedia: Metaprogramming.
`http://en.wikipedia.org/wiki/metaprogramming_(programming)`.
Stand: 28.01.2005.

[187] X. Ye. On the relationship between finite volume and finite element methods applied to the Stokes equations. *Numer. Methods Partial Differential Equations*, 17(5):440–453, 2001.

[188] O. C. Zienkiewicz. Origins, milestones and directions of the finite element method—a personal view. In *Handbook of numerical analysis, Vol. IV*, pages 3–67. North-Holland, Amsterdam, 1996.

[189] O. C. Zienkiewicz and O. nate E. Finite volumes vs Finite elements. Is there really a choice? In W. P. and W. W., editors, *Nonlinear computational mechanics*, pages 240–254. Springer-Verlag, Berlin, 1991.

[190] O. Zwintzscher. *Komponentenbasierte & generative Software-Entwicklung.* W3L-Verlag, Herdecke, 2003.

Anhang

A.1 Exakte Integration

Alle hier angegebenen Integrale wurden mit Hilfe von *Mathematica* berechnet und in einzelnen Fällen analytisch und numerisch mit *Matlab* geprüft.

$A_1(a-bc,b)$
$A_2(0,0)$
$A_3(a,0)$
$A_{12}(\frac{a-bc}{2},\frac{b}{2})$
$A_{13}(a-\frac{bc}{2},\frac{b}{2})$
$A_{23}(\frac{a}{2},0)$

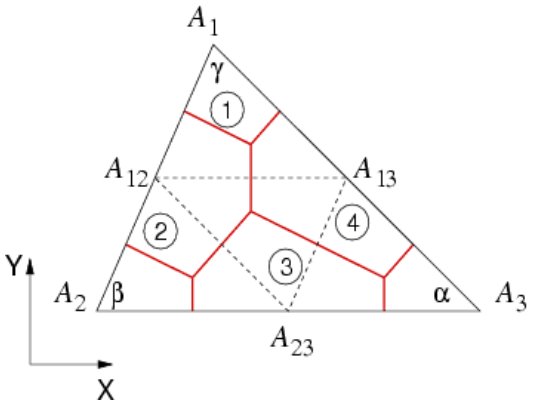


Abbildung A.1: Ein verfeinertes Dreieck mit dem dualen Voronoi-Diagramm

P_1^M/P_0-Element

Die Geschwindigkeits-Ansatzfunktionen φ_j^i des Knotens A_j im Dreieck K_i:

$$\begin{aligned}
\varphi_1^1 &= \tfrac{2}{b}y-1, & \varphi_{12}^1 &= -\tfrac{2}{a}x-\tfrac{2c}{a}y+2,\\
\varphi_2^2 &= -\tfrac{2}{a}x-\tfrac{2c}{a}y+1, & \varphi_{12}^2 &= \tfrac{2}{b}y,\\
\varphi_3^4 &= \tfrac{2}{a}x+\left(\tfrac{2c}{a}-\tfrac{2}{b}\right)y-1, & \varphi_{12}^3 &= -\tfrac{2}{a}x+\left(-\tfrac{2c}{a}+\tfrac{2}{b}\right)y+1,\\
\varphi_{13}^1 &= \tfrac{2}{a}x+\left(\tfrac{2c}{a}-\tfrac{2}{b}\right)y, & \varphi_{23}^2 &= \tfrac{2}{a}x+\left(\tfrac{2c}{a}-\tfrac{2}{b}\right)y,\\
\varphi_{13}^3 &= \tfrac{2}{a}x+\tfrac{2c}{a}y-1, & \varphi_{23}^3 &= -\tfrac{2}{b}y+1,\\
\varphi_{13}^4 &= \tfrac{2}{b}y, & \varphi_{23}^4 &= -\tfrac{2}{a}x-\tfrac{2c}{a}y+2,
\end{aligned} \tag{A.1}$$

die auf den in (A.1) nicht genannten Dreiecken verschwinden. Die konstante Druck-Ansatzfunktion ist $\psi_K = 1$.

Ergänzung für die Formeln (3.44)-(3.45) der Integration der Druckterme:

$$\begin{cases} \displaystyle\int_K \frac{\partial\varphi_2}{\partial x}\,dxdy\,p^K = -\frac{b}{4}\,p^K & = (-m^K_{2,12}\cos\beta - m^K_{2,23})\,p^K \\ & = (m^K_{2,12}\tilde{n}^x_{2,12} + m^K_{2,23}\tilde{n}^x_{2,23})\,p^K \\ \displaystyle\int_K \frac{\partial\varphi_2}{\partial y}\,dxdy\,p^K = -\frac{bc}{4}\,p^K & = (-m^K_{2,12}\sin\beta)\,p^K \\ & = -(m^K_{2,12}\tilde{n}^y_{2,12} + m^K_{2,23}\tilde{n}^y_{2,23})\,p^K \end{cases} \tag{A.2}$$

$$\begin{cases} \displaystyle\int_K \frac{\partial\varphi_3}{\partial x}\,dxdy\,p^K = \frac{b}{4}\,p^K & = (m^K_{3,13}\cos\alpha + m^K_{3,23})\,p^K \\ & = -(m^K_{3,13}\tilde{n}^x_{3,23} + m^K_{3,23}\tilde{n}^x_{3,23})\,p^K \\ \displaystyle\int_K \frac{\partial\varphi_3}{\partial y}\,dxdy\,p^K = -\frac{a-bc}{4}\,p^K & = (-m^K_{3,13}\sin\alpha)\,p^K \\ & = -(m^K_{3,13}\tilde{n}^y_{3,13} + m^K_{3,23}\tilde{n}^y_{3,23})\,p^K \end{cases} \tag{A.3}$$

$$\begin{cases} \displaystyle\int_K \frac{\partial\varphi_{13}}{\partial x}\,dxdy\,p^K = \frac{b}{2}p^K = & (m_{13,23}\cos\beta + m_{13,12})\,p^K \\ & = -(m_{13,23}\tilde{n}^x_{13,23} + m_{13,12}\tilde{n}^x_{13,12} \\ & \quad + m^K_{13,1}\tilde{n}^x_{13,1} + m^K_{13,3}\tilde{n}^x_{13,3})\,p^K, \\ \displaystyle\int_K \frac{\partial\varphi_{13}}{\partial y}\,dxdy\,p^K = \frac{bc}{2}p^K = & (m_{13,23}\sin\beta)\,p^K \\ & = -(m_{13,23}\tilde{n}^y_{13,23} + m_{13,12}\tilde{n}^y_{13,12} \\ & \quad + m^K_{13,1}\tilde{n}^y_{13,1} + m^K_{13,3}\tilde{n}^y_{13,3})\,p^K. \end{cases} \tag{A.4}$$

$$\begin{cases} \displaystyle\int_K \frac{\partial\varphi_{23}}{\partial x}\,dxdy\,p^K = 0p^K = & (m_{23,12}\cos\alpha - m_{23,13}\cos\beta + m^K_{23,2} - m^K_{23,3})\,p^K \\ & = -(m_{23,12}\tilde{n}^x_{23,12} + m_{23,13}\tilde{n}^x_{23,13} \\ & \quad + m^K_{23,2}\tilde{n}^x_{23,2} + m^K_{23,3}\tilde{n}^x_{23,3})\,p^K, \\ \displaystyle\int_K \frac{\partial\varphi_{23}}{\partial y}\,dxdy\,p^K = -\frac{a}{2}p^K = & -(m_{23,12}\sin\alpha + m_{23,13}\sin\beta)\,p^K \\ & = -(m_{23,12}\tilde{n}^y_{23,12} + m_{23,13}\tilde{n}^y_{23,13} \\ & \quad + m^K_{23,2}\tilde{n}^y_{23,2} + m^K_{23,3}\tilde{n}^y_{23,3})\,p^K. \end{cases} \tag{A.5}$$

Die Approximation der Kontinuitätsgleichung ist vollständig in (3.46) angegeben.

P_1^M/P_1-Element

Die Geschwindigkeits-Ansatzfunktionen sind in (A.1) dargestellt. Für die Druck-Ansatzfunktionen gilt

$$\begin{aligned} \psi_1^K &= \tfrac{1}{b}y, \\ \psi_2^K &= -\tfrac{1}{a}x - \tfrac{c}{a}y + 1, \\ \psi_3^K &= \tfrac{1}{a}x + \left(\tfrac{c}{a} - \tfrac{1}{b}\right)y + 1. \end{aligned} \tag{A.6}$$

Ergänzung für die Formeln (3.48)-(3.49) der Integration der Druckterme:

$$\begin{cases} \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_2}{\partial x} dxdy\, p_j = -\frac{b}{24}p_1 - \frac{b}{6}p_2 - \frac{b}{24}p_3 = \frac{p_1+4p_2+p_3}{6}(-m^K_{2,23} - m^K_{2,12}\cos\beta), \\ \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_2}{\partial y} dxdy\, p_j = -\frac{bc}{24}p_1 - \frac{bc}{6}p_2 - \frac{bc}{24}p_3 = \frac{p_1+4p_2+p_3}{6}(-m^K_{2,12}\sin\beta). \end{cases} \tag{A.7}$$

$$\begin{cases} \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_3}{\partial x} dxdy\, p_j = \frac{b}{24}p_1 + \frac{b}{24}p_2 + \frac{b}{6}p_3 = \frac{p_1+p_2+4p_3}{6}(m^K_{3,23} + m^K_{3,13}\cos\alpha), \\ \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_3}{\partial y} dxdy\, p_j = -\frac{a-bc}{24}p_1 - \frac{a-bc}{24}p_2 - \frac{a-bc}{6}p_3 = \frac{p_1+p_2+4p_3}{6}(-m^K_{3,13}\sin\alpha). \end{cases} \tag{A.8}$$

$$\begin{cases} \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_{13}}{\partial x} dxdy\, p_j &= \frac{b}{4}p_1 + \frac{b}{8}p_2 + \frac{b}{8}p_3 \\ &= \frac{4p_1+p_2+p_3}{6}m^K_{1,13}\cos\beta + \frac{2p_1+p_2+p_3}{4}m_{13,12} \\ &\quad + \frac{p_1+p_2+2p_3}{4}m_{13,23}\cos\alpha - \frac{p_1+p_2+4p_3}{6}m^K_{3,13}\cos\beta, \\ \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_{13}}{\partial y} dxdy\, p_j &= -\frac{a-2bc}{8}p_1 + \frac{bc}{8}p_2 + \frac{a+bc}{8}p_3 \\ &= -\frac{4p_1+p_2+p_3}{6}m^K_{1,13}\sin\beta + \frac{p_1+p_2+2p_3}{4}m_{13,23}\sin\alpha \\ &\quad + \frac{p_1+p_2+4p_3}{6}m^K_{3,13}\sin\beta. \end{cases} \tag{A.9}$$

$$\begin{cases} \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_{23}}{\partial x} dxdy\, p_j &= 0p_1 + \frac{b}{8}p_2 - \frac{b}{8}p_3 \\ &= \frac{p_1+2p_2+p_3}{4}m_{12,23}\cos\beta - \frac{p_1+p_2+2p_3}{4}m^K_{13,23}\cos\alpha \\ &\quad + \frac{p_1+4p_2+p_3}{6}m^K_{2,23} - \frac{p_1+p_2+4p_3}{6}m^K_{3,23}, \\ \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_{23}}{\partial y} dxdy\, p_j &= -\frac{a}{8}p_1 - \frac{2a-bc}{8}p_2 - \frac{a+bc}{8}p_3 \\ &= -\frac{p_1+2p_2+p_3}{4}m_{12,23}\sin\beta - \frac{p_1+p_2+2p_3}{4}m^K_{13,23}\sin\alpha. \end{cases} \tag{A.10}$$

Kontinuitätsgleichung:

$$\begin{aligned}
\sum_{i\in I}\int_K \psi_1\left(\frac{\partial\varphi_i}{\partial x}u_i+\frac{\partial\varphi_i}{\partial y}v_i\right)dxdy &= 0u_1-\tfrac{b}{24}u_2+\tfrac{b}{24}u_3-\tfrac{b}{4}u_{12}+\tfrac{b}{4}u_{13}+0u_{23}\\
&\quad+\tfrac{a}{6}v_1-\tfrac{bc}{24}v_2-\tfrac{a-bc}{24}v_3+\tfrac{a-2bc}{8}v_{12}-\tfrac{a-2bc}{8}v_{13}-\tfrac{a}{8}v_{23}\\
\sum_{i\in I}\int_K \psi_2\left(\frac{\partial\varphi_i}{\partial x}u_i+\frac{\partial\varphi_i}{\partial y}v_i\right)dxdy &= 0u_1-\tfrac{b}{6}u_2+\tfrac{b}{24}u_3-\tfrac{b}{8}u_{12}+\tfrac{b}{8}u_{13}+\tfrac{b}{8}u_{23}\\
&\quad+\tfrac{a}{24}v_1-\tfrac{bc}{6}v_2-\tfrac{a-bc}{24}v_3+\tfrac{2a-bc}{8}v_{12}-\tfrac{bc}{8}v_{13}-\tfrac{2a-bc}{8}v_{23}\\
\sum_{i\in I}\int_K \psi_3\left(\frac{\partial\varphi_i}{\partial x}u_i+\frac{\partial\varphi_i}{\partial y}v_i\right)dxdy &= 0u_1-\tfrac{b}{24}u_2+\tfrac{b}{6}u_3-\tfrac{b}{8}u_{12}+\tfrac{b}{8}u_{13}-\tfrac{b}{8}u_{23}\\
&\quad+\tfrac{a}{24}v_1-\tfrac{bc}{24}v_2-\tfrac{a-bc}{6}v_3+\tfrac{a-bc}{8}v_{12}-\tfrac{a+bc}{8}v_{13}-\tfrac{a+bc}{8}v_{23}
\end{aligned} \tag{A.11}$$

P_2/P_0-Element

Die Geschwindigkeits-Ansatzfunktionen zweiter Ordnung:

$$\begin{aligned}
\varphi_1^K &= \tfrac{2}{b^2}y^2-\tfrac{1}{b}y,\\
\varphi_2^K &= \tfrac{2}{a^2}x^2+\tfrac{4c}{a^2}xy+\tfrac{2c^2}{a^2}y^2-\tfrac{3}{a}x-\tfrac{3c}{a}y+1,\\
\varphi_3^K &= \tfrac{2}{a^2}x^2+\left(\tfrac{4c}{a^2}-\tfrac{4}{ab}\right)xy+\left(\tfrac{2c^2}{a^2}-\tfrac{4c}{ab}+\tfrac{2}{b^2}\right)y^2-\tfrac{1}{a}x+\left(\tfrac{1}{b}-\tfrac{c}{a}\right)y,\\
\varphi_{12}^K &= -\tfrac{4}{ab}xy-\tfrac{4c}{ab}y^2+\tfrac{4}{b}y,\\
\varphi_{13}^K &= \tfrac{4}{ab}xy+\left(\tfrac{4c}{ab}-\tfrac{4}{b^2}\right)y^2,\\
\varphi_{23}^K &= -\tfrac{4}{a^2}x^2+\left(-\tfrac{8c}{a^2}+\tfrac{4}{ab}\right)xy+\left(-\tfrac{4c^2}{a^2}+\tfrac{4c}{ab}\right)y^2+\tfrac{4}{a}x+\left(\tfrac{4c}{a}-\tfrac{4}{b}\right)y.
\end{aligned} \tag{A.12}$$

Die konstante Druck-Ansatzfunktion $\psi_K = 1$.
Ergänzung für die Formeln (3.56)-(3.57) der Integration der Diffusionsterme:

$$\begin{aligned}
\sum_{i\in I}\int_K \nabla\varphi_i\nabla\varphi_2 d\boldsymbol{x}\,\boldsymbol{u}_i &= \tfrac{c}{6}\boldsymbol{u}_1+\tfrac{b(1+c^2)}{2a}\boldsymbol{u}_2+\tfrac{b(1+c^2)-ac}{2a}\boldsymbol{u}_3-\tfrac{2c}{3}\boldsymbol{u}_{12}+\tfrac{2ac-2b(1+c^2)}{3a}\boldsymbol{u}_{23}\\
&= \left(\tfrac{1}{6}\boldsymbol{u}_1+\tfrac{1}{2}\boldsymbol{u}_2-\tfrac{2}{3}\boldsymbol{u}_{12}\right)\cot\alpha+\left(\tfrac{1}{2}\boldsymbol{u}_2+\tfrac{1}{6}\boldsymbol{u}_3-\tfrac{2}{3}\boldsymbol{u}_{23}\right)\cot\gamma\\
&= \tfrac{2(3\boldsymbol{u}_2-4\boldsymbol{u}_{12}+\boldsymbol{u}_1)}{3a_{2,12}}m_{2,12}^K+\tfrac{2(3\boldsymbol{u}_2-4\boldsymbol{u}_{23}+\boldsymbol{u}_3)}{3a_{2,23}}m_{2,23}^K,
\end{aligned} \tag{A.13}$$

$$\begin{aligned}
\sum_{i\in I}\int_K \nabla\varphi_i\nabla\varphi_3 d\boldsymbol{x}\,\boldsymbol{u}_i &= \tfrac{a-bc}{6b}\boldsymbol{u}_1+\tfrac{b(1+c^2)-ac}{6a}\boldsymbol{u}_2+\left(\tfrac{a}{2b}+\tfrac{b(1+c^2)}{2a}-c\right)\boldsymbol{u}_3\\
&\quad-\tfrac{2(a-bc)}{3b}\boldsymbol{u}_{13}+\tfrac{2ac-2b(1+c^2)}{3a}\boldsymbol{u}_{23}\\
&= \left(\tfrac{1}{2}\boldsymbol{u}_3+\tfrac{1}{6}\boldsymbol{u}_1-\tfrac{2}{3}\boldsymbol{u}_{13}\right)\cot\beta+\left(\tfrac{1}{2}\boldsymbol{u}_3+\tfrac{1}{6}\boldsymbol{u}_2-\tfrac{2}{3}\boldsymbol{u}_{23}\right)\cot\gamma\\
&= \tfrac{2(3\boldsymbol{u}_3-4\boldsymbol{u}_{13}+\boldsymbol{u}_1)}{3a_{3,13}}m_{3,13}^K+\tfrac{2(3\boldsymbol{u}_3-4\boldsymbol{u}_{23}+\boldsymbol{u}_2)}{3a_{3,23}}m_{3,23}^K,
\end{aligned} \tag{A.14}$$

$$
\begin{aligned}
&\sum_{i\in I}\int_K \nabla\varphi_i\nabla\varphi_{13}d\boldsymbol{x}\,\boldsymbol{u}_i = \\
&= -\tfrac{2(a-bc)}{3b}\boldsymbol{u}_1 - \tfrac{2(a-bc)}{3b}\boldsymbol{u}_3 + \tfrac{4}{3}\left(c - \tfrac{b(1+c^2)}{a}\right)\boldsymbol{u}_{12} + \tfrac{4}{3}\left(\tfrac{a}{b} + \tfrac{b(1+c^2)}{a} - c\right)\boldsymbol{u}_{13} - \tfrac{4c}{3}\boldsymbol{u}_{23} \\
&= -\tfrac{2}{3}(\boldsymbol{u}_1+\boldsymbol{u}_3)\cot\beta - \tfrac{4}{3}\boldsymbol{u}_{23}\cot\alpha - \tfrac{4}{3}\boldsymbol{u}_{12}\cot\gamma + \tfrac{4}{3}(\cot\alpha+\cot\beta+\cot\gamma)\boldsymbol{u}_{13} \\
&= \tfrac{4(\boldsymbol{u}_{13}-\boldsymbol{u}_1)}{3a_{1,13}}m^K_{1,13} + \tfrac{4(\boldsymbol{u}_{13}-\boldsymbol{u}_3)}{3a_{3,13}}m^K_{3,13} + \tfrac{4(\boldsymbol{u}_{13}-\boldsymbol{u}_{12})}{3a_{13,12}}m_{13,12} + \tfrac{4(\boldsymbol{u}_{13}-\boldsymbol{u}_{23})}{3a_{23,13}}m_{23,13},
\end{aligned}
\tag{A.15}
$$

$$
\begin{aligned}
&\sum_{i\in I}\int_K \nabla\varphi_i\nabla\varphi_{23}d\boldsymbol{x}\,\boldsymbol{u}_i = \\
&= \tfrac{2}{3}\left(c - \tfrac{b(1+c^2)}{a}\right)\boldsymbol{u}_2 + \tfrac{2}{3}\left(c - \tfrac{b(1+c^2)}{a}\right)\boldsymbol{u}_3 - \tfrac{4(a-bc)}{3b}\boldsymbol{u}_{12} - \tfrac{4c}{3}\boldsymbol{u}_{13} + \tfrac{4}{3}\left(\tfrac{a}{b} + \tfrac{b(1+c^2)}{a} - c\right)\boldsymbol{u}_{23} \\
&= -\tfrac{2}{3}(\boldsymbol{u}_2+\boldsymbol{u}_3)\cot\gamma - \tfrac{4}{3}\boldsymbol{u}_{12}\cot\beta - \tfrac{4}{3}\boldsymbol{u}_{13}\cot\alpha + \tfrac{4}{3}(\cot\alpha+\cot\beta+\cot\gamma)\boldsymbol{u}_{12} \\
&= \tfrac{4(\boldsymbol{u}_{23}-\boldsymbol{u}_2)}{3a_{2,23}}m^K_{2,23} + \tfrac{4(\boldsymbol{u}_{23}-\boldsymbol{u}_3)}{3a_{3,23}}m^K_{3,23} + \tfrac{4(\boldsymbol{u}_{23}-\boldsymbol{u}_{12})}{3a_{23,12}}m_{23,12} + \tfrac{4(\boldsymbol{u}_{23}-\boldsymbol{u}_{13})}{3a_{23,13}}m_{23,13},
\end{aligned}
\tag{A.16}
$$

Ergänzung für die Formeln (3.58)-(3.59) der Integration der Druckterme:

$$
\begin{cases}
\displaystyle\int_K \frac{\partial\varphi_2}{\partial x}\,dxdy\,p^K = -\frac{b}{6}\,p^K &= \tfrac{2}{3}(m^K_{2,12}\tilde{n}^x_{2,12} + m^K_{2,23}\tilde{n}^x_{2,23})\,p^K \\
\displaystyle\int_K \frac{\partial\varphi_2}{\partial y}\,dxdy\,p^K = -\frac{bc}{6}\,p^K &= \tfrac{2}{3}(m^K_{2,12}\tilde{n}^y_{2,12} + m^K_{2,23}\tilde{n}^y_{2,23})\,p^K,
\end{cases}
\tag{A.17}
$$

$$
\begin{cases}
\displaystyle\int_K \frac{\partial\varphi_3}{\partial x}\,dxdy\,p^K = \frac{b}{6}\,p^K &= \tfrac{2}{3}(m^K_{3,13}\tilde{n}^x_{3,13} + m^K_{3,23}\tilde{n}^x_{3,23})\,p^K \\
\displaystyle\int_K \frac{\partial\varphi_3}{\partial y}\,dxdy\,p^K = -\frac{a-bc}{6}\,p^K &= \tfrac{2}{3}(m^K_{3,13}\tilde{n}^y_{3,13} + m^K_{3,23}\tilde{n}^y_{3,23})\,p^K,
\end{cases}
\tag{A.18}
$$

$$
\begin{cases}
\displaystyle\int_K \frac{\partial\varphi_{13}}{\partial x}\,dxdy\,p^K = \frac{2b}{3}\,p^K = & \tfrac{4}{3}(m_{13,23}\tilde{n}^x_{13,23} + m_{13,12}\tilde{n}^x_{13,12} \\
& +m^K_{13,1}\tilde{n}^x_{13,1} + m^K_{13,2}\tilde{n}^x_{13,2})\,p^K, \\
\displaystyle\int_K \frac{\partial\varphi_{13}}{\partial y}\,dxdy\,p^K = \frac{2bc}{3}\,p^K = & \tfrac{4}{3}(m_{13,23}\tilde{n}^y_{13,23} + m_{13,12}\tilde{n}^y_{13,12} \\
& +m^K_{13,1}\tilde{n}^y_{13,1} + m^K_{13,2}\tilde{n}^y_{13,2})\,p^K.
\end{cases}
\tag{A.19}
$$

$$
\begin{cases}
\displaystyle\int_K \frac{\partial\varphi_{23}}{\partial x}\,dxdy\,p^K = 0\,p^K = & \tfrac{4}{3}(m_{23,12}\tilde{n}^x_{23,12} + m_{23,13}\tilde{n}^x_{23,13} \\
& +m^K_{23,2}\tilde{n}^x_{23,2} + m^K_{23,3}\tilde{n}^x_{23,3})\,p^K, \\
\displaystyle\int_K \frac{\partial\varphi_{23}}{\partial y}\,dxdy\,p^K = -\frac{2a}{3}\,p^K = & \tfrac{4}{3}(m_{23,12}\tilde{n}^y_{23,12} + m_{23,13}\tilde{n}^y_{23,13} \\
& +m^K_{23,2}\tilde{n}^y_{23,2} + m^K_{23,3}\tilde{n}^y_{23,3})\,p^K.
\end{cases}
\tag{A.20}
$$

Approximation der Kontinuitätsgleichung ist vollständig in (3.60) angegeben.

P_2/P_1-Taylor-Hood-Element

Druckterme:

$$\begin{cases} \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_1}{\partial x} dxdy\, p_j = 0 = \frac{2}{3} p_1 (m_{1,12}^K \cos\beta - m_{1,13}^K \cos\alpha), \\ \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_1}{\partial y} dxdy\, p_j = \frac{a}{6} p_1 = \frac{2}{3} p_1 (m_{1,12}^K \sin\beta + m_{1,13}^K \sin\alpha), \end{cases} \tag{A.21}$$

$$\begin{cases} \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_2}{\partial x} dxdy\, p_j = -\frac{b}{6} p_2 = -\frac{2}{3} p_2 (m_{2,12}^K \cos\beta + m_{2,23}^K), \\ \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_2}{\partial y} dxdy\, p_j = -\frac{bc}{6} p_2 = -\frac{2}{3} p_2\, m_{2,12}^K \sin\beta, \end{cases} \tag{A.22}$$

$$\begin{cases} \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_3}{\partial x} dxdy\, p_j = \frac{b}{6} p_3 = \frac{2}{3} p_3 (m_{3,13}^K \cos\alpha + m_{3,23}^K), \\ \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_3}{\partial y} dxdy\, p_j = -\frac{a-bc}{6} p_3 = -\frac{2}{3} p_3\, m_{3,13}^K \sin\alpha, \end{cases} \tag{A.23}$$

$$\begin{cases} \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_{12}}{\partial x} dxdy\, p_j &= -\frac{b}{3} p_1 - \frac{b}{6} p_2 - \frac{b}{6} p_3 \\ &= -\frac{2}{3} p_1 m_{1,12}^K \cos\beta + \frac{2}{3} p_2 m_{2,12}^K \cos\beta \\ &\quad - \frac{2p_1+p_2+p_3}{3} m_{12,13} - \frac{p_1+2p_2+p_3}{3} m_{12,23} \cos\alpha, \\ \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_{12}}{\partial y} dxdy\, p_j &= \frac{a-2bc}{6} p_1 + \frac{2a-bc}{6} p_2 + \frac{a-bc}{6} p_3 \\ &= -\frac{2}{3} p_1 m_{1,12}^K \sin\beta + \frac{2}{3} p_2 m_{2,12}^K \sin\beta \\ &\quad + \frac{p_1+2p_2+p_3}{3} m_{12,23} \sin\alpha, \end{cases} \tag{A.24}$$

$$\begin{cases} \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_{13}}{\partial x} dxdy\, p_j &= \frac{b}{3} p_1 + \frac{b}{6} p_2 + \frac{b}{6} p_3 \\ &= \frac{2}{3} p_1 m_{1,13}^K \cos\alpha - \frac{2}{3} p_3 m_{3,13}^K \cos\alpha \\ &\quad + \frac{2p_1+p_2+p_3}{3} m_{13,12} + \frac{p_1+p_2+2p_3}{3} m_{13,23} \cos\beta, \\ \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_{13}}{\partial y} dxdy\, p_j &= -\frac{a-2bc}{6} p_1 - \frac{bc}{6} p_2 + \frac{a+bc}{6} p_3 \\ &= -\frac{2}{3} p_1 m_{1,13}^K \sin\alpha + \frac{2}{3} p_2 m_{3,13}^K \sin\alpha \\ &\quad + \frac{p_1+p_2+2p_3}{3} m_{13,23} \sin\beta, \end{cases} \tag{A.25}$$

$$\begin{cases} \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_{23}}{\partial x} dxdy\, p_j &= 0p_1 + \frac{b}{6}p_2 - \frac{b}{6}p_3 = \frac{2}{3}p_2 m^K_{2,23} - \frac{2}{3}p_3 m^K_{3,23} \\ & \quad + \frac{p_1+2p_2+p_3}{3} m_{12,23}\cos\alpha - \frac{p_1+p_2+2p_3}{3} m_{13,23}\cos\beta, \\ \sum_{j=1}^{3} \int_K \psi_j \frac{\partial \varphi_{23}}{\partial y} dxdy\, p_j &= -\frac{a}{6}p_1 - \frac{2a-bc}{6}p_2 - \frac{a+bc}{6}p_3 \\ & = -\frac{p_1+2p_2+p_3}{3} m_{12,23}\sin\alpha - \frac{p_1+p_2+2p_3}{3} m_{13,23}\sin\beta, \end{cases} \tag{A.26}$$

Kontinuitätsgleichung:

$$\begin{aligned} \sum_{i\in I} \int_K \psi_1 \left(\nabla\varphi_i\, \boldsymbol{u}_i\right) d\boldsymbol{x} &= -\tfrac{b}{3}u_{12} + \tfrac{b}{3}u_{13} + \tfrac{a}{6}v_1 + \tfrac{a-2bc}{6}v_{12} - \tfrac{a-2bc}{6}v_{13} - \tfrac{a}{6}v_{23} \\ &= \left(\tfrac{a_{12}}{6}\sin\beta - \tfrac{a_{13}}{6}\sin\alpha\right) u_1 - \tfrac{a_{12}}{3}\sin\beta\, u_{12} + \tfrac{a_{13}}{3}\sin\alpha\, u_{13} \\ + \left(\tfrac{a_{12}}{6}\cos\beta + \tfrac{a_{13}}{6}\cos\alpha\right) v_1 &+ \left(\tfrac{a_{12}}{3}\cos\beta\right) v_{12} + \left(\tfrac{a_{13}}{3}\cos\alpha\right) v_{13} - \tfrac{a}{6}\left(v_{12}+v_{13}+v_{23}\right), \\ \sum_{i\in I} \int_K \psi_2 \left(\nabla\varphi_i\, \boldsymbol{u}_i\right) d\boldsymbol{x} &= -\tfrac{b}{6}u_2 - \tfrac{b}{6}u_{12} + \tfrac{b}{6}u_{13} + \tfrac{b}{6}u_{23} \\ &\quad -\tfrac{bc}{6}v_2 + \tfrac{2a-bc}{6}v_{12} + \tfrac{bc}{6}v_{13} - \tfrac{2a-bc}{6}v_{23} \\ \sum_{i\in I} \int_K \psi_3 \left(\nabla\varphi_i\, \boldsymbol{u}_i\right) d\boldsymbol{x} &= \tfrac{b}{6}u_3 - \tfrac{b}{6}u_{12} + \tfrac{b}{6}u_{13} - \tfrac{b}{6}u_{23} \\ &\quad -\tfrac{a-bc}{6}v_3 + \tfrac{a-bc}{6}v_{12} + \tfrac{a+bc}{6}v_{13} - \tfrac{a+bc}{6}v_{23} \end{aligned} \tag{A.27}$$

A.2 Diagramm-Notation

A.2.1 Merkmaldiagramme

Ein Konzept wird durch eine Menge von Merkmalen (engl. features) beschrieben. Jedes dieser Merkmale kann wiederum durch andere Untermerkmale näher beschrieben werden. So entsteht eine Hierarchie, die in einem Merkmaldiagramm dargestellt wird. Die Wurzel eines Merkmaldiagramms repräsentiert das Konzept selbst und wird als Konzeptknoten bezeichnet. Dieser Knoten ist immer in einem Merkmaldiagramm enthalten. Die restlichen Knoten repräsentieren Merkmale und heißen Eigenschaftsknoten. Tabelle A.1 fasst kurz Elemente eines Merkmaldiagramms (engl. Feature Diagram) zusammen, die von Czarnecki und Eisenecker [38] systematisiert worden sind und in Kapitel 4 dieser Arbeit benutzt werden.

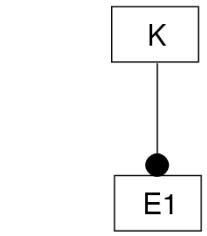
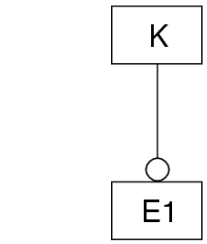
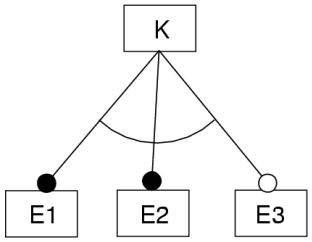
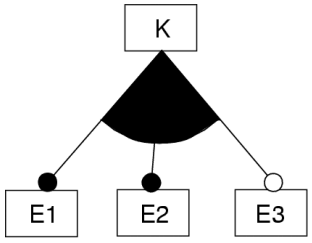
Diagramm-Beispiel	Beschreibung
	festes Merkmal (engl. feature): Das Konzept K enthält das Merkmal E1
	optionales Merkmal: Das Konzept K kann das Merkmal E1 enthalten
	alternative Merkmale: Das Konzept K enthält ein einziges Merkmal von E1, E2, E3
	oder-Merkmale: Das Konzept K enthält mindestens ein Merkmal von E1, E2, E3

Tabelle A.1: Notation eines Merkmaldiagramms

A.2.2 UML-Diagramme

Die UML-Sprache (engl. Unified Modeling Language) ist eine Sprache zur Definition, Visualisierung, Konstruktion und Dokumentierung von Softwaresysteme. Sie benutzt folgende Arten von Diagrammen: Use-Case-Diagramme, Klassen-Diagramme, Verhalten-Diagramme und Implementierungs-Diagramme. In dieser Arbeit (Abb. 5.3) werden nur die Klassen-Diagramme benutzt, deren Notation kurz in der Tabelle A.2 zusammengefasst ist. Die vollständige Beschreibung findet man in der UML-Spezifikation [164].

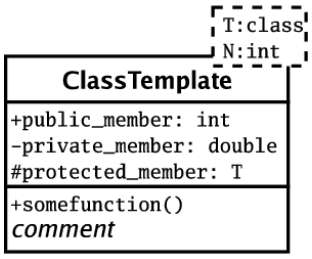
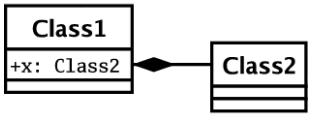
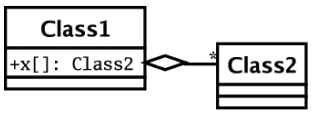
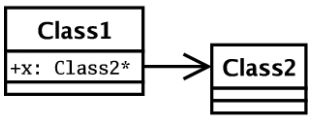
Diagramm-Beispiel	Beschreibung
	Definition des Klassen-Templates `ClassTemplate` mit den Parameter `T` als Datentyp und `N` als ganze Zahl. In der ersten Sektion unter dem Namen werden die Daten-Mitglieder (auch Attribute) definiert. Die zweite Sektion enthält die Klassen-Funktionen.
	Komposition: die Klasse `Class1` enthält ein Objekt der Klasse `Class2`.
	Aggregation: die Klasse `Class1` enthält mehrere Objekte der Klasse `Class2`, z.B. in einem Array.
	Assoziation: die Klasse `Class1` enthält eine Referenz auf ein Objekt der Klasse `Class2`.

Tabelle A.2: Notation eines UML-Klassendiagramms

A.3 Lineare-Algebra-Benchmark-Tests

Diese Benchmark-Tests wurden mit dem Ziel durchgeführt, die Skalarprodukt-Operation mit der Unrolling-Strategie mittels der Template-Metaprogrammierung zu untersuchen. Im Abschnitt 5.3.3 wurden die Voraussetzungen für die Tests sowie die Resultaten für einen Pentium4-Prozessor bereits präsentiert. Nach der Definition der normalisierten MFLOPS werden hier die mit unterschiedlichen Compiler übersetzte Benchmark-Tests auf drei weiteren Hardware-Plattformen analysiert.

A.3.1 Normalisierte MFLOPS

Um die Benchmark-Tests auf unterschiedlichen Hardware-Plattformen und für verschiedene Vektorlänge vergleichen zu können, wird die Rechenleistung in normalisierten MFLOPS gemessen, da nicht alle mathematischen Operationen eine Gleitkomma-Operation beinhalten. Der Term 'normalisiert' bedeutet, dass jeder Operationstyp ein Gewicht bekommt. Diese Vorgehensweise wurde im bekannten Livermore-Loop-Benchmark [100] benutzt, wobei für Addition, Subtrahieren und Multiplikation 1, für Division und Quadratwurzel 4 und für trigonometrische Funktionen der Faktor 8 auf allen Hardware-Plattformen angenommen wurde. Das ist offensichtlich nur eine Näherungsannahme. Die exakten Werte können mit einem Extra-Test [118] gewonnen werden. Die Ergebnisse dieses Tests befinden sich in der Tabelle A.3. Die dabei begleitende theoretische Höchstleistung bedeutet die oberste nicht erreichbare Prozessorleistung, die als die Taktfrequenz multipliziert mit der Anzahl der Prozessor-Recheneinheiten berechnet wird. Eine wirkliche Leistung wird häufig in einer Prozentzahl von diesem Wert angegeben.

Hardware-Plattform	Betriebssystem	Peak Performance (GFLOPS)	MFLOPS-Normalisierungsfaktor			
			add	mul	div	sin
Pentium III 800MHz	RedHat Linux 9 (Kern 2.4.20)	0.8	1	1	1.6	21.6
Pentium4 2,4GHz	Fedora Core 2 (Kern 2.6.6)	4.8	1	1	1.2	21.4
AMD Opteron 848 (2,4GHz)	Fedora Core 2 (Kern 2.6.5-1)	4.8	1	1	1	3.7
IBM Power4 1,5Ghz	IBM AIX 5.2	6.0	1	1	5.1	15.2

Tabelle A.3: Hardware-Beschreibung zu Benchmark-Tests

A.3.2 AMD Opteron 848

Der AMD-Opteron-Prozessor hat die besondere Eigenschaft, gleichzeitig sowohl 32- als auch 64-bit-Anwendungen ausführen zu können. Für die Benchmark-Tests auf diesem Prozessor wurden drei Compiler benutzt (Abb. A.2): GNU C++ 3.3.3, GNU C++ 2.96 und Intel C++ 8.1, wobei nur der erste 64-bit-Architektur unterstützt. Er hat auch die besten Ergebnisse gezeigt. Der für eine andere Plattform gedachte Intel-Compiler zeigt vergleichbar gute Ergebnisse. Die Zusammenfassung der Tests

wird in der Form von Beschleunigungs-Koeffizienten bezüglich des gewöhnlichen Schleifen-Codes dargestellt (Abb. A.2,a). Wie erwartet ergeben sich die größten Beschleunigungs-Koeffizienten für $N = 2, 4, 8$. Zum Vergleich werden auch dieselben Tests mit der prozessoroptimierten ACML-Bibliothek und der Fortran-Sprache durchgeführt. Die Tests mit größten Koeffizienten sind ausführlich abhängig von der Vektorlänge auf den weiteren Grafiken angegeben (Abb. A.2,b,c,d). Genau wie auf Pentium4-Prozessor ist der CPU-Zeitgewinn nur für mittellange Vektoren deutlich. Auch für ganz kleine Vektoren (Abb. A.2,e,f) gibt es einen klaren Gewinn.

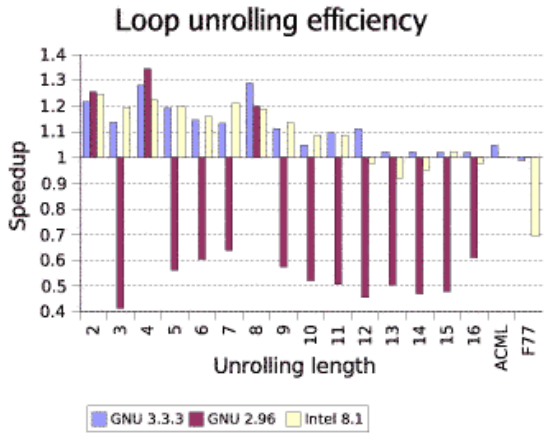
A.3.3 Pentium III 800MHz

Der PentiumIII-Prozessor mit nur einer Rechen-Einheit zeigt ganz negative Ergebnisse, wobei nur für $N = 2$ ein kleiner Vorteil zu sehen ist (Abb. A.3). Dagegen erzeugt der Fortran-Compiler einen besseren Code.

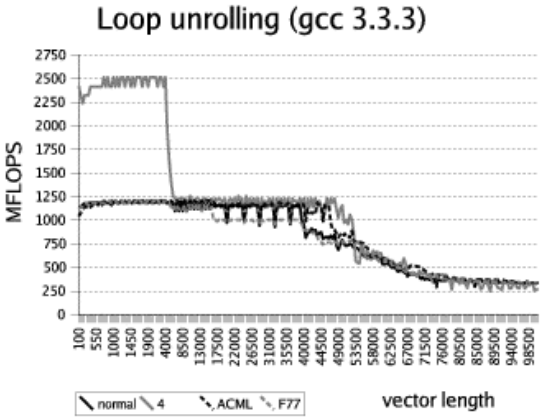
A.3.4 IBM Power4

Der IBM-Power4-Prozessor verfügt über die größte Leistung mit 4 Rechen-Einheiten bei niedrigerer Taktfrequenz und erzielt die höchsten Beschleunigungs-Koeffizienten (Abb. A.4,a). Alle Beschleunigungs-Koeffizienten sind positiv und erreichen die dreifache Leistung, obwohl die gesamtgrößte Leistung mit beiden verwendeten Compiler (IBM und GNU C++) mit der hardwareoptimierten IBM-ESSL-Bibliothek gezeigt wurde.

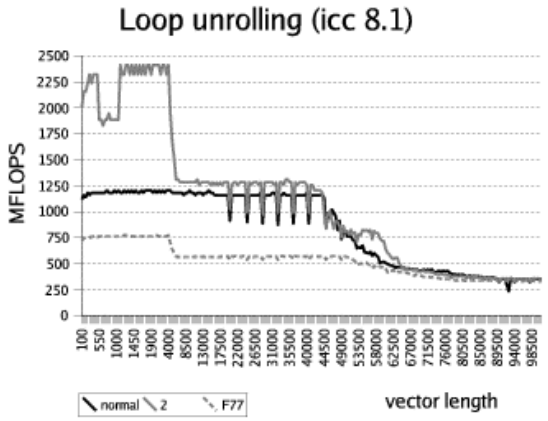
Im Unterschied zu allen vorhergehenden Tests gibt es keinen großen Stufen-Sprung im Leistungsverlauf (Abb. A.4,c,d). Das kann man durch einen Level-III-Cachespeicher (128Mb) erklären, den dieser Prozessor besitzt.



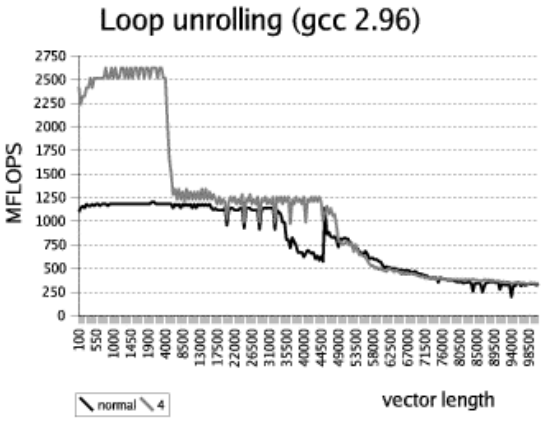
a) Beschleunigungs-Koeffizienten



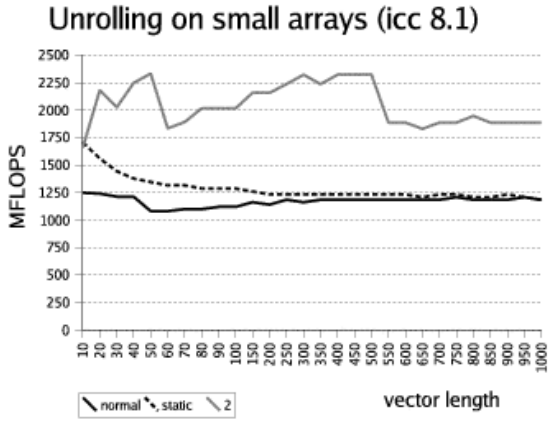
b) GNU gcc 3.3.3 Compiler



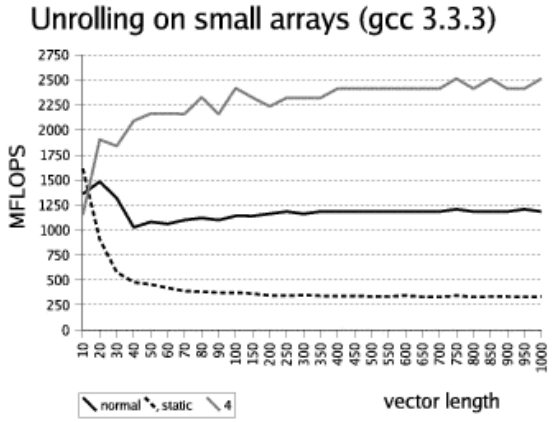
c) Intel icc 8.1 Compiler



d) GNU gcc 2.96 Compiler

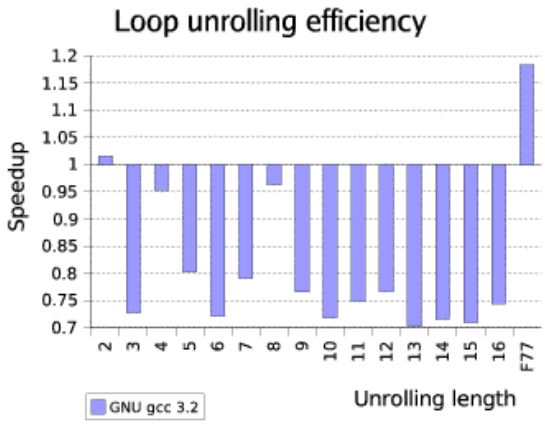


e) Kleine Arrays (icc 8.1)

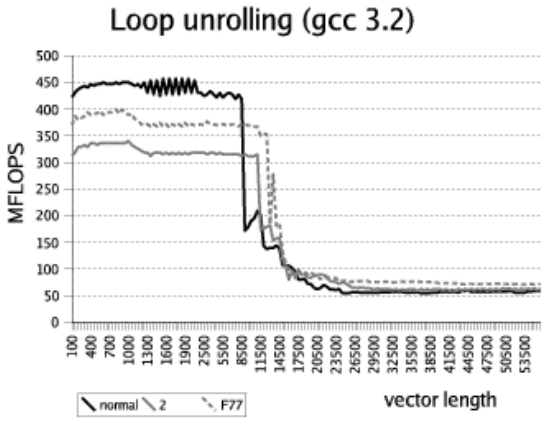


f) Kleine Arrays (gcc 3.3.3)

Abbildung A.2: Loop-Unrolling durch Expression-Templates auf einem AMD-Opteron-Prozessor

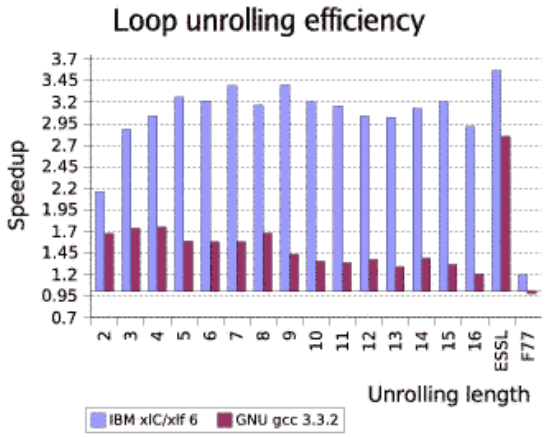


a) Beschleunigungs-Koeffizienten

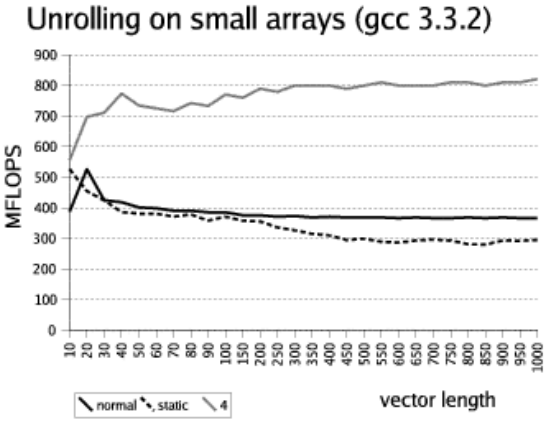


b) GNU gcc 2.96 Compiler

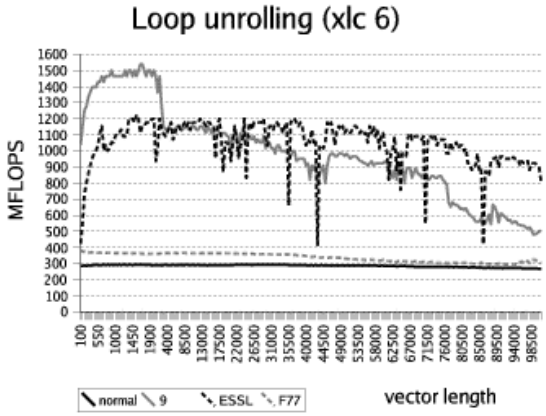
Abbildung A.3: Loop-Unrolling durch Expression-Templates auf einem Intel-PentiumIII-Prozessor



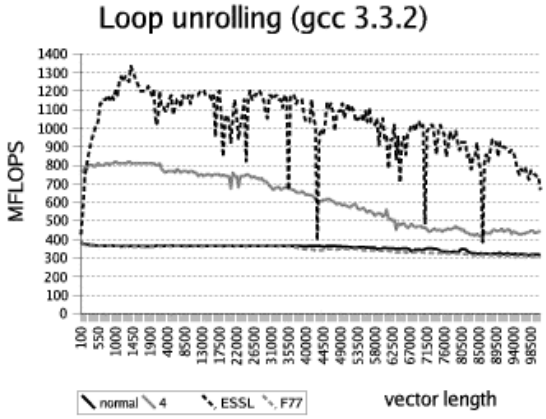
a) Beschleunigungs-Koeffizienten



b) Kleine Arrays



c) IBM xlc 6 Compiler



d) GNU gcc 3.3.3 Compiler

Abbildung A.4: Loop-Unrolling durch Expression-Templates auf einem IBM-Power4-Prozessor

A.4 Meta-Programme

A.4.1 Aufbau von Ansatzfunktionen

Hier werden die im Abschnitt 4.3 beschriebenen Algorithmen zum Aufbau von polynomialen Ansatzfunktionen als Meta-Programmen dargestellt. Ein Monom wird durch eine Numliste repräsentiert (Listing A.1), wobei der Koeffizient und die gleiche Potenz für alle Variablen im Aufbauprogramm `Monomial` angegeben werden können. Eine weitere Bearbeitung von Monomen erfolgt mit Hilfe von allgemeinen Meta-Algorithmen für die Numlisten (Tab. 4.5). So wird im Meta-Programm `BasicTerm` (Listing A.2) eine bestimmte Potenz des Monoms mit Hilfe des Programms `AddAt` auf 1 gesetzt. Das Meta-Programm `BasicTerm` läuft rekursiv über alle Variablen $x_d, ..., x_1$ und liefert das Polynom $-mx_d - ... - mx_1$ in der Form einer Typliste.

```
1 // Dim - dimension
  // Coef - integer coefficient
3 // Power - default power for all variables
  template<unsigned short Dim,
5          int Coef=0, int Power=0>
  struct Monomial
7 {
    typedef Numlist<Coef,
9           typename InitNumlist<Dim,Power>::Result> Result;
  };
```

Listing A.1: Meta-Programm: Definition eines Monoms

```
  // Dim - dimension
2 // Order - polynomial order
  // Num - counter (must not be defined)
4 template<unsigned short Dim, unsigned short Order,
          unsigned short Num=Dim>
6 struct BasicTerm
  {
8   typedef Typelist<typename AddAt<
            typename Monomial<Dim,-Order>::Result,Num,1>::Result,
10          typename BasicTerm<Dim,Order,Num-1>::Result> Result;
  };
12 template<unsigned short Dim, unsigned short Order>
  struct BasicTerm<Dim,Order,0>
14 {
    typedef NullType Result;
16 };
```

Listing A.2: Meta-Programm: Erzeugung des Polynoms $(-mx_d - ... - mx_1)$

Im Meta-Programm `BasicList3` (Listing A.3) werden bestimmte Konstanten zu diesem Polynom addiert und die resultierende Polynome miltipliziert. Das Produkt von linearen Polynomen wird durch eine Typliste gebildet, wobei die Klasse

`UnitType` statt `NullType` im Unterschied zu einem Polynom in Standardform als Endmarke benutzt wird.

```
// Num - number of linear polynomials in the product
template<unsigned short Dim, unsigned short Order,
         unsigned short Num>
struct BasicList3 {
  typedef Typelist<
           Typelist<typename Monomial<Dim,Order-Num+1>::Result,
                    typename BasicTerm<Dim,Order>::Result>,
           typename BasicList3<Dim,Order,Num-1>::Result> Result;
};
template<unsigned short Dim, unsigned short Order>
struct BasicList3<Dim,Order,0> {
   typedef UnitType Result;
};
```

Listing A.3: Meta-Programm: Erzeugung des Produkts der linearen Polynome $(m - mx_d - ... - mx_1)...(m - k + 1 - mx_d - ... - mx_1)$

```
template<unsigned short Dim, unsigned short Order,
         unsigned short I, unsigned short Num>
struct ShapeBasis {
private:
   typedef typename NL::AddAt<
      typename Monomial<Dim,Order>::Result,I,1>::Result T1;
   typedef typename Monomial<Dim,1-Num>::Result T2;
public:
   typedef Typelist<TYPELIST_2(T1,T2),
           typename ShapeBasis<Dim,Order,I,Num-1>
                                  ::Result> Result;
};
template<unsigned short Dim, unsigned short Order,
         unsigned short I>
struct ShapeBasis<Dim,Order,I,0> {
   typedef UnitType Result;
};
```

Listing A.4: Meta-Programm: Erzeugung vom Produkt der linearen Polynome $(mx_i - k)...mx_i$

Ähnlicherweise bildet das Meta-Programm `ShapeBasis` (Listing A.4) ein Produkt der linearen Polynome $(mx_i - k)...mx_i$.

Das Meta-Programm `ShapeFunction3` (Listing A.5) bekommt eine d-dimensionale Nummer `NList` der Ansatzfunktion in der Form einer Numliste, läuft diese Liste durch und ruft mit dem entsprechenden Parameter `Num` das Meta-Programm `BasicList3` auf. Die restlichen bis zu m Stück linearer Polynome in der Produktform werden vom Meta-Programm `ShapeBasis` geliefert. Außer dem Typ `Result`,

der die Typliste zu Darstellung der endgültigen Ansatzfunktion auf einem Dreieckelement liefert, berechnet das Meta-Programm `ShapeFunction3` mit Hilfe von `Factorial` auch eine ganze Zahl `denom`, die der Nenner bzw. der Skalierungsparameter der Ansatzfunktion ist.

```
// NList - Dim-dimensional number of the shape function
// N - internal counter of the NList number
// Sum - internal sum of the NList-components
template<unsigned short Dim, unsigned short Order, class NList,
         unsigned short N=1, unsigned short Sum=0>
struct ShapeFunction3;
template<unsigned short Dim, unsigned short Order,
         unsigned short N, unsigned short Sum>
struct ShapeFunction3<Dim,Order,NL::NullType,N,Sum>
{
   enum { denom = Factorial<Order-Sum>::value };
   typedef typename BasicList3<Dim,Order,Order-Sum>
                                        ::Result Result;
};
template<unsigned short Dim, unsigned short Order,
    int Num, class Tail, unsigned short N, unsigned short Sum>
struct ShapeFunction3<Dim,Order,Numlist<Num,Tail>,N,Sum>
{
private:
   typedef ShapeFunction3<Dim,Order,Tail,N+1,Sum+Num> temp;
public:
   enum { denom = Factorial<Num>::value * temp::denom };
   typedef typename TL::Append<
            typename ShapeBasis<Dim,Order,N,Num>::Result,
            typename temp::Result>::Result Result;
};
```

Listing A.5: Meta-Programm: Erzeugung einer Ansatzfuntion auf einem Dreieckelement

Hilfsprogramme: Multiplikation und Vereinfachung

Damit eine polynomiale Ansatzfunktion weiter behandelt (z.B. differenziert oder integriert) werden kann, muss die durch das Meta-Programm `ShapeFunction3` (Listing A.5) erhaltene Produktform in die Polynom-Standardform transformiert werden, d.h. alle linearen Polynome des Produktes müssen multipliziert werden. Das dafür entwickelte Meta-Programm `Mult` bekommt zwei zu multiplizierende Objekte. Das können zwei Monome, zwei Polynome in der Standardform oder ein Monom und ein Polynom sein (Listing A.6). Weitere Fälle sind im Listing A.7 spezialisiert, wobei nun auch Polynome in der Produktform multipliziert werden können. Zur Abkürzung werden hier einige offensichtliche Teile der Implementierung durch Kommentare ersetzt. Bei der Multiplikation von Polynomen entstehen oft ähnliche Glieder, die mit dem Meta-Programm `Simplify` (Listing A.8) zusammengesetzt werden können.

```
template<class List1, class List2> struct Mult;
// 1) simple implementation of the operations:
//Mult<NullType,NullType> -> NullType
//Mult<NullType,Numlist> -> NullType
//Mult<Numlist,NullType> -> NullType
//Mult<NullType,Typelist> -> NullType
//Mult<Typelist,NullType> -> NullType
// 2) monomial by monomial multiplication:
template<int Coef1, class Tail1, int Coef2, class Tail2>
struct Mult<Numlist<Coef1,Tail1>,Numlist<Coef2,Tail2>>
{
   typedef Typelist<Numlist<Coef1*Coef2,
           typename Add<Tail1,Tail2>::Result>,NullType> Result;
};
// 3) monomial by polynomial multiplication:
template<int C1, class T1, int C2, class T2, class Tail>
struct Mult<Numlist<C1,T1>, Typelist<Numlist<C2,T2>,Tail>>
{
   typedef typename TL::Append<
           typename Mult<Numlist<C1,T1>,
                         Numlist<C2,T2> >::Result,
           typename Mult<Numlist<C1,T1>,Tail>::Result
                                 >::Result Result;
};
// 4) polynomial by monomial multiplication:
//the same specialization with exchanged template parameters,
//i.e. Mult<Typelist<Numlist<C1,T1>,Tail>,Numlist<C2,T2>>
// 5) polynomial by polynomial multiplication:
template<int C1, class T1, class Tail1,
         int C2, class T2, class Tail2>
struct Mult<Typelist<Numlist<C1,T1>,Tail1>,
            Typelist<Numlist<C2,T2>,Tail2>>
{
private:
   typedef Numlist<C1,T1> Head1;
   typedef Numlist<C2,T2> Head2;
public:
   typedef typename TL::Append<
           typename TL::Append<
           typename TL::Append<
           typename Mult<Head1,Head2>::Result,
           typename Mult<Head1,Tail2>::Result>::Result,
           typename Mult<Tail1,Head2>::Result>::Result,
           typename Mult<Tail1,Tail2>::Result>::Result Result;
};
```

Listing A.6: Meta-Programm: Multiplikation von Monomen und Polynomen in Standardform

```
// 6) multiplications by 1:
//Mult<UnitType, UnitType> -> UnitType
//Mult<UnitType, Typelist> -> Typelist
//Mult<Typelist, UnitType> -> Typelist
// 7) polynomial in standard form by polynomial in product form:
template<int C, class T1, class Tail1,
         class H, class T2, class Tail2>
struct Mult<Typelist<Numlist<C,T1>,Tail1>,
            Typelist<Typelist<H,T2>,Tail2> >
{
private:
   typedef typename Mult<Typelist<H,T2>,Tail2>::Result temp;
public:
   typedef typename TL::Append<
            typename Mult<Numlist<C,T1>,temp>::Result,
            typename Mult<Tail1,temp>::Result>::Result Result;
};
// 8) polynomial in product form by polynomial in standard form:
//the same specialization with exchanged template parameters
//i.e. Mult<Typelist<Typelist<..>,T1>,Typelist<Numlist<..>,T2>>
// 9) polynomial in product form by polynomial in product form:
template<class H1, class T1, class Tail1,
         class H2, class T2, class Tail2>
struct Mult<Typelist<Typelist<H1,T1>,Tail1>,
            Typelist<Typelist<H2,T2>,Tail2> >
{
   typedef typename Mult<
        typename Mult<Typelist<H1,T1>,Typelist<H2,T2> >::Result,
        typename Mult<Tail1,Tail2>::Result>::Result Result;
};

//Transformation of a polynomial TList from the product
//into the standard form
template<class TList> struct Expand;

template<class H, class T, class Tail>
struct Expand<Typelist<Typelist<H,T>,Tail> >
{
   typedef typename Mult<Typelist<H,T>,Tail>::Result Result;
};
```

Listing A.7: Meta-Programm: Multiplikation von Polynomen in Standard- und Produktform

```
template<class TList, class Term> struct SumAndDel;
template<class Term>
struct SumAndDel<NullType,Term>
{
   enum { sum = 0 };
   typedef NullType Result;
};
template<int Coef, class Term, class Tail>
struct SumAndDel<Typelist<Numlist<Coef,Term>,Tail>,Term>
{
private:
   typedef SumAndDel<Tail,Term> temp;
public:
   enum { sum = Coef + temp::sum };
   typedef typename temp::Result Result;
};
template<class Head, class Tail, class Term>
struct SumAndDel<Typelist<Head,Tail>,Term>
{
private:
   typedef SumAndDel<Tail,Term> temp;
public:
   enum { sum = temp::sum };
   typedef Typelist<Head,typename temp::Result> Result;
};

template<class TList> struct Simplify;
template<>
struct Simplify<NullType>
{
   typedef NullType Result;
};
template<int Coef, class T, class Tail>
struct Simplify<Typelist<Numlist<Coef,T>,Tail> >
{
private:
   typedef SumAndDel<Tail,T> temp;
   enum { newcoef = Coef + temp::sum };
   typedef typename Simplify<
            typename temp::Result>::Result newlist;
public:
   typedef typename Loki::Select<newcoef==0,newlist,
        Typelist<Numlist<newcoef,T>,newlist> >::Result Result;
};
```

Listing A.8: Meta-Programm: Mathematische Vereinfachung eines Polynoms in Standardform

Differenziation und Integration

Polynome in Standardform können einfach differenziert werden. Das Meta-Programm `Differentiate` (Listing A.9) differenziert jedes Monom eines Polynoms nach der Regel

$$\frac{\partial}{\partial x_i} c x_1^{p_1} ... x_d^{p_d} = c p_i x_1^{p_1} ... x_i^{p_i - 1} ... x_d^{p_d} \tag{A.28}$$

mit Hilfe der Algorithmen `AddAt` und `MultAt` über Numlisten. Das Meta-Programm nutzt den Algorithmus `Select<T,T0,T1>` der Bibliothek *Loki* [3], der als Resultat `T0` liefert, falls `T` erfüllt ist und `T1` anderenfalls. Dadurch wird ein Monom weggelassen, falls es keine Variable x_i enthält. Ansonsten wird die Regel (A.28) verwendet.

Das Meta-Programm `Integrate3` (Listing A.10) läuft alle Monome eines in `TList` gegebenen Polynoms durch und benutzt ein Hilfsprogramm `IntegrateTerm3`, das ein Monom nach der Formel (4.8) über einem Dreieckelement integriert. Das Resultat gibt die statische Funktion `val()` zurück.

```
// TList - a polynomial in the standard form
// Var - number of the variable (1..Dim)
template<class TList, unsigned short Var> struct Differentiate;

template<unsigned short Var>
struct Differentiate<NullType,Var>
{
   typedef NullType Result;
};
template<class NList, class Tail, unsigned short Var>
struct Differentiate<Typelist<NList,Tail>,Var>
{
private:
   enum { pow = NL::NumAt<NList,Var>::value };
   typedef typename Differentiate<Tail,Var>::Result next;
public:
   typedef typename Loki::Select<pow==0,next,
         Typelist<typename NL::AddAt<
                  typename NL::MultAt<NList,0,pow>
               ::Result,Var,-1>::Result,next> >::Result Result;
};
```

Listing A.9: Meta-Programm: Partielle Differenziation eines Polynoms

```
template<class NList, unsigned short Len=0>
struct IntegrateTerm3;

template<unsigned short Len>
struct IntegrateTerm3<NL::NullType,Len>
{
   enum { numerator=1, denominator=Len };
};
template<int Num, class Tail, unsigned short Len>
struct IntegrateTerm3<Numlist<Num, Tail>,Len>
{
private:
   typedef IntegrateTerm3<Tail,Len+1> temp;
public:
   enum { numerator = Factorial<Num>::value * temp::numerator,
          denominator = Num + temp::denominator };
};

template<class TList> struct Integrate3;
template<>
struct Integrate3<Loki::NullType>
{
   inline static double val()  {  return 0; }
};
template<int Num, class T, class Tail>
struct Integrate3<Typelist<Numlist<Num,T>,Tail> >
{
private:
   typedef IntegrateTerm3<T> iterm;
public:
   inline static double val()
   {  return Num*iterm::numerator/static_cast<double>(
             Factorial<iterm::denominator>::value)
             + Integrate3<Tail>::val();
   }
};
```

Listing A.10: Meta-Programm: Integration eines Polynoms

A.4.2 Loop-Unrolling

Das Loop-Unrolling mittels Template-Metaprogrammierung zählt zu den Performance-Optimierungsverfahren auf dem unteren Niveau und wurde in den Tests der linearen Algebra im Abschnitt 5.3.3 benutzt. Die Idee besteht in der Einteilung eines Array in kleine Abschnitte mit einer festen und vor der Compilierung angegebenen Länge N, die dann durch Metaprogramme statt üblicher Schleifen behandelt werden. Das Loop-Unrolling wird im Klassen-Template `Unroller` (Listing A.11) implementiert, das die Länge N als Template-Parameter definiert. Nach diesem Parameter werden die statischen Funktionen (hier als Beispiel `assign, add` und `dot`) rekursiv aufgerufen bis die Schlussspezialisierung `Unroller<1>` erreicht ist. Dort wird die Operation mit dem letzten Element durchgeführt.

```
template<unsigned int N>
struct Unroller {
    // assignment of two arrays
    inline static void assign(double* a1, double* a2) {
        *a1 = *a2;
        Unroller<N-1>::assign(a1+1,a2+1);
    }
    // addition of two arrays
    inline static void add(double* a1, double* a2) {
        *a1 += *a2;
        Unroller<N-1>::add(a1+1,a2+1);
    }
    // dot product
    inline static double dot(double* a1, double* a2) {
        return (*a1 * *a2) + Unroller<N-1>::dot(a1+1,a2+1);
    }
};

template<>
struct Unroller<1> {
    inline static void assign(double* a1, double* a2) {
        *a1 = *a2;
    }
    inline static void add(double* a1, double* a2) {
        *a1 += *a2;
    }
    inline static double dot(double* a1, double* a2) {
        return *a1 * *a2;
    }
};
```

Listing A.11: Loop-Unrolling durch Template-Metaprogrammierung

A.5 Simulations-Ergebnisse

A.5.1 Zylinderumströmung in 2D

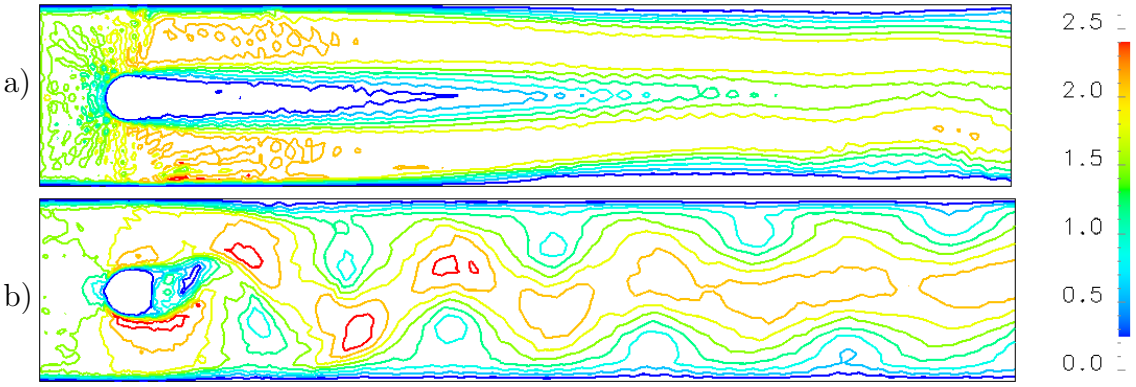


Abbildung A.5: Geschwindigkeits-Konturen der Zylinderumströmung, diskretisiert mit dem Crank-Nicolson/Semi-Explizit-Schema: a) P_2/P_0-, b) P_2/P_1-Stokes-Element; $t = 3.0s$ in beiden Fällen

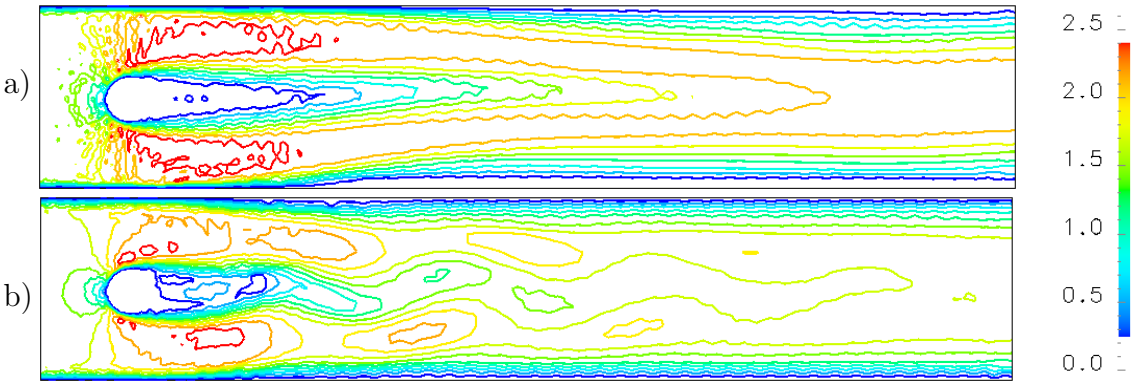
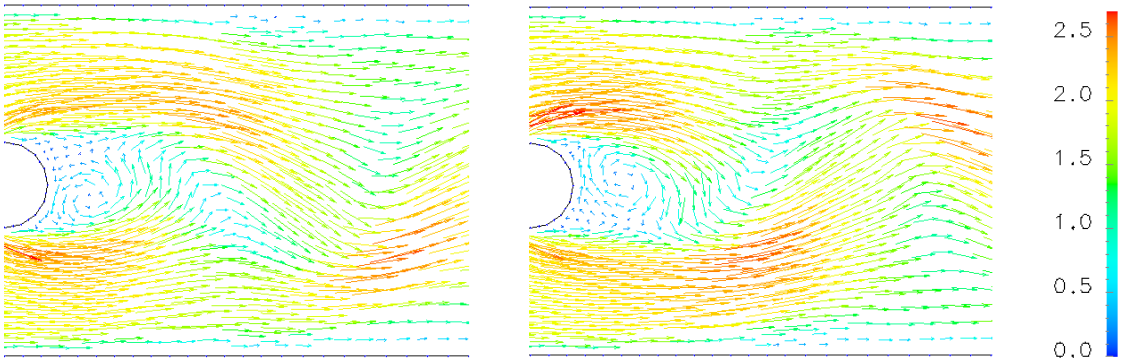


Abbildung A.6: Geschwindigkeits-Konturen der Zylinderumströmung, diskretisiert mit dem Euler/Newton-Schema: a) P_2/P_0, $t = 1.16s$; b) P_2/P_1, $t = 3.0s$



$t = 3.6s$ $t = 3.75s$

Abbildung A.7: Geschwindigkeits-Vektoren der Zylinderumströmung, diskretisiert mit dem Crank-Nicolson/Newton-Schema und P_2/P_1-Element

A.5.2 Komplexe Umströmung in 3D

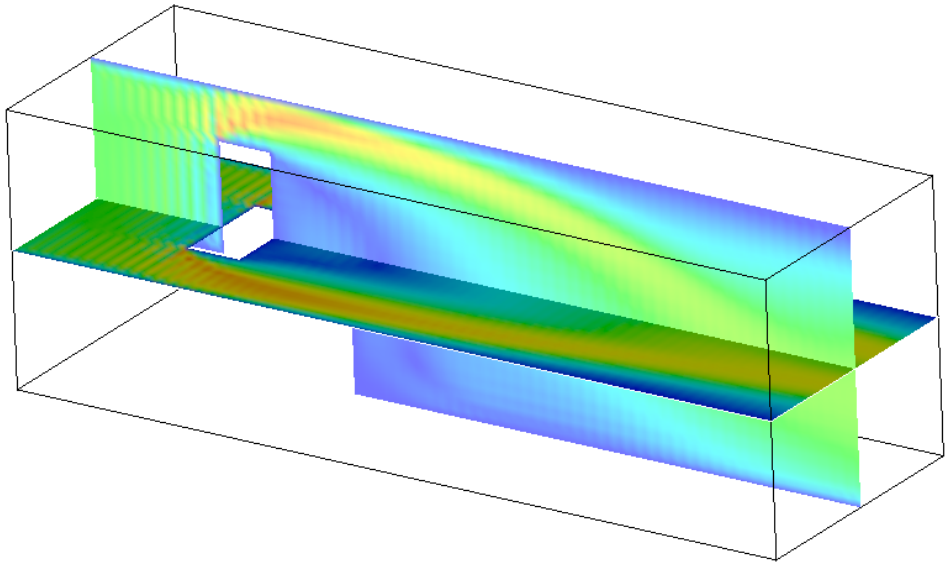


Abbildung A.8: Umströmung eines Hindernisses in 3D: Felder des Geschwindigkeits-Betrages, Q_2/Q_0-Element, $t = 4.49s$

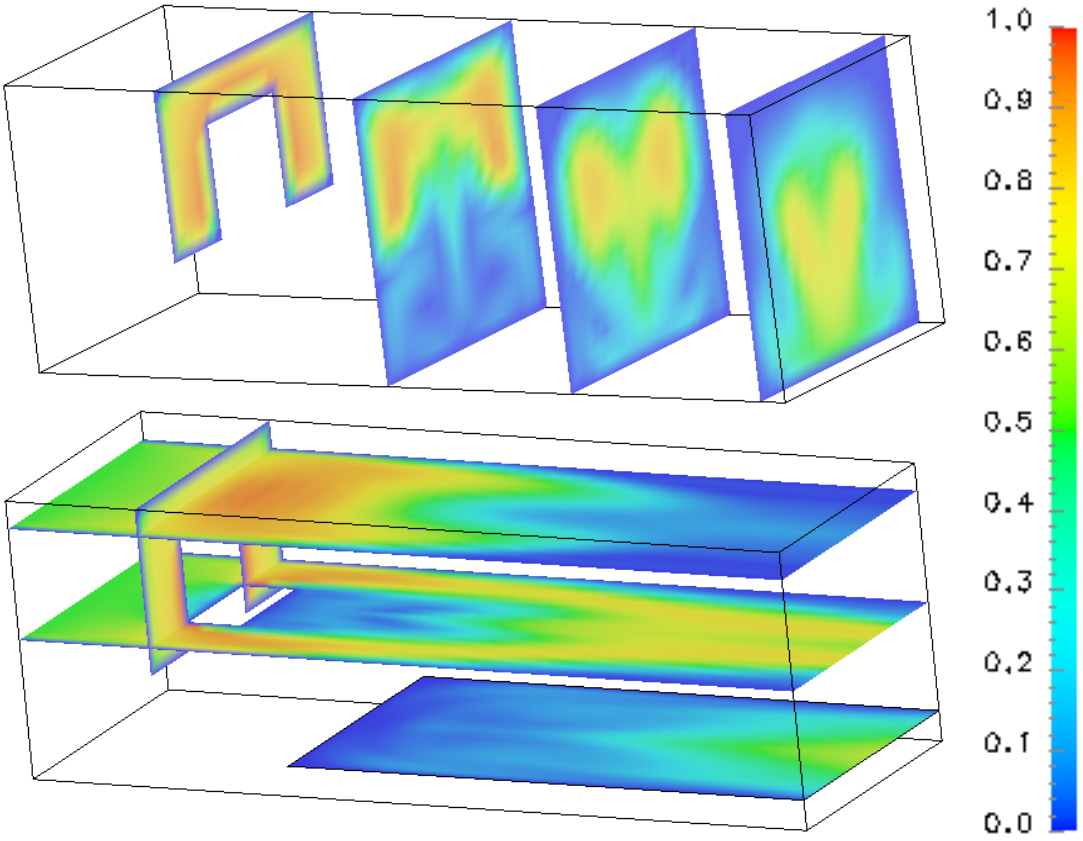
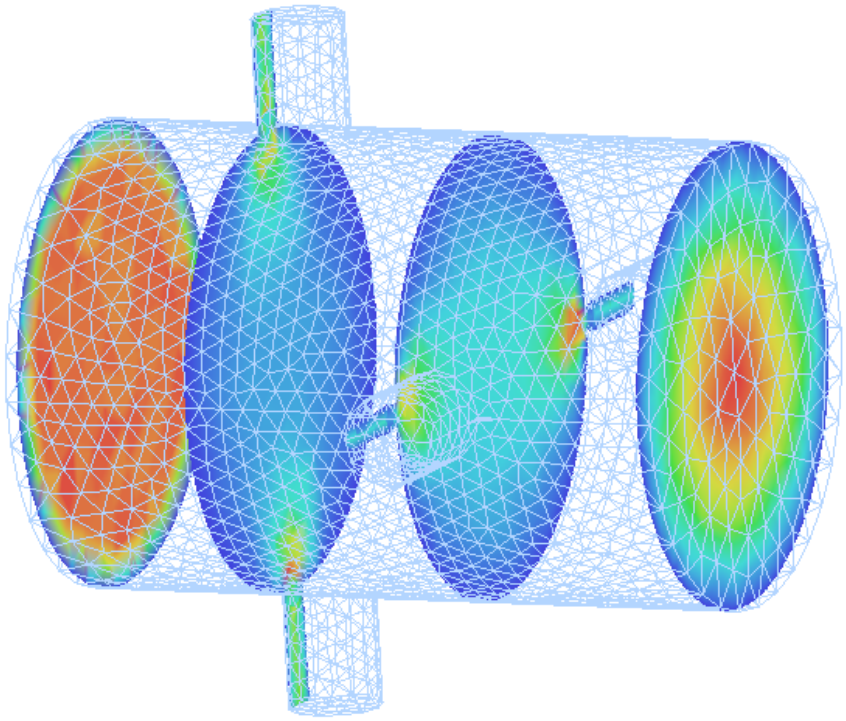
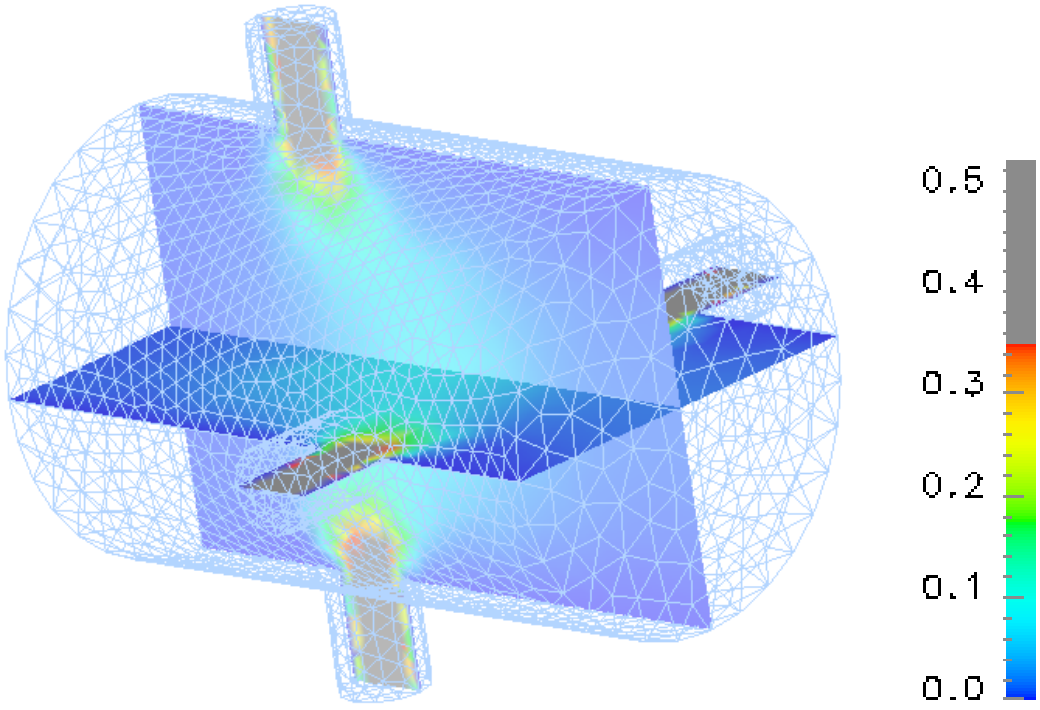


Abbildung A.9: Umströmung eines Hindernisses in 3D: Felder des Geschwindigkeits-Betrages, Q_2/Q_1-Element, $t = 5.25s$

A.5.3 Eine Pipeline in 3D



a) yz-Querschnitte (man beachte, dass die Schnitte unterschiedliche Skalen aufweisen)



b) xy- und xz-Querschnitte

Abbildung A.10: Geschwindigkeitsfeld in yz-Querschnitten einer Stokes-Strömung in der Pipeline, diskretisiert mit Euler/Semi-Explizit-Schema und P_2/P_1-Element auf einem Tetraeder-Gitter

Symbolverzeichnis

Römische Zeichen

A, $\tilde{A}$	Steifigkeitsmatrix (Abschnitt 5.2.4)
A_i	Knoten eines Gitters (Def. 3.2), Ecken eines allgemeinen Dreiecks (Abb. 3.6,3.7)
A_{ij}	Mittelpunkt der Kante A_iA_j (Abb. 3.7)
A_V	Voronoi-Punkt (Abschnitt 3.2)
B	Matrix des negativen diskreten Divergenz-Operators
B_3	Raum der kubischen Blasen-Funktionen (3.66)
C, C_i	eine Konstante
C_p	Druck-Massenmatrix (Abschnitt 5.2.6)
D	Block eines Block-Präkonditionierers (Abschnitt 5.2.6)
$\mathcal{E}_h$	Menge aller inneren Kanten eines Gitters (Abschnitt 5.2.8)
F	Matrix des diskreten Konvektions-Diffusions-Operators
G	Block eines Block-Präkonditionierers (Abschnitt 5.2.6)
H	Block eines Block-Präkonditionierers (Abschnitt 5.2.6)
K, K_i	Zelle eines Gitters
K_Φ	Komplexität der Berechnung der Transformationsdaten (Abschnitt 5.2.4)
K_i	Koinzidenzmatrix des Teilgitters i (Abschnitt 5.2.7)
L, L_{ij}	Lokale Steifigkeitsmatrix (Abschnitt 5.2.3)
L_k	ein lineares Polynom (Abschnitt 4.3)
M	Dimension der lokalen Steifigkeitsmatrix bzw. Anzahl der Freiheitsgrade auf einem Gitter-Element (Abschnitt 4.4)
Ma	Mach-Zahl
M_e	Speicherplatz für eine Gitter-Entity (Abschnitt 5.2.1)
M_c	Speicherplatz für eine Gitter-Entity-Beziehung (Abschnitt 5.2.1)
M_u	Anzahl der Geschwindigkeits-Freiheitsgrade auf einem Gitter-Element (Abschnitt 4.4,5.2.3)
M_p	Anzahl der Druck-Freiheitsgrade auf einem Gitter-Element (Abschnitt 4.4,5.2.3)
N_K	Anzahl der Gitterknoten
N_R	Anzahl der Gitter-Randknoten
N_D	Anzahl der Dirichlet-Randknoten
N_T	Anzahl der Gitterzellen
N_u	Anzahl der Geschwindigkeits-Freiheitsgrade eines Gitters
N_p	Anzahl der Druck-Freiheitsgrade eines Gitters

$\mathcal{N}_I$	Punkte bzw. Freiheitsgrade im Teilgebietsinneren (Abschnitt 5.2.7)
$\mathcal{N}_E$	Punkte im Inneren der Teilgebietskanten (Abschnitt 5.2.7)
$\mathcal{N}_V$	Punkte am Beginn oder am Ende einer Teilgebietskante (Abschnitt 5.2.7)
P	Block-Präkonditionierer (Abschnitt 5.2.6)
P_k	Menge der Polynome der Ordnung nicht größer als k bezüglich aller Variablen
Q_k	Menge der Polynome der Ordnung nicht größer als k bezüglich jeder Variable
Re	Reynolds-Zahl
S	Matrix des Schur-Komplement-Operators (Abschnitt 5.2.6)
T	Temperatur (Kap. 2)
$\mathcal{Q}_h$	Ein Viereckgitter
$\mathcal{T}$, $\mathcal{T}_h$	Ein Gitter oder ein Dreieckgitter
$\mathcal{V}$	Voronoi-Diagramm
$\mathcal{V}_i$	Voronoi-Polygon
W_k	Wertigkeit eines Gitter-Elements (Abschnitt 5.2.7)
X, X_M, X_B X_R, X_G	Block eines Block-Präkonditionierers (Abschnitt 5.2.6)
a	Schallgeschwindigkeit (Abschnitt 2.1)
a_{ij}	Länge einer Dreieckseite (Abschnitt 3.5)
$a(\cdot,\cdot)$, $b(\cdot,\cdot)$	Bilinearformen (Abschnitt 3.4)
$c(\cdot,\cdot,\cdot)$	Trilinearform (Abschnitt 3.4)
a, b, c	Basis auf einem Dreieck (Abschnitt 3.5)
b_{jl}	Komponente der Matrix $(\Phi\Phi^T)^{-1}$ (Abschnitt 5.2.3)
c_{gl}	Komponente der Matrix Φ^{-1} (Abschnitt 5.2.3)
c_p	isobare spezifische Wärmekapazität (Kap. 2)
d	Raumdimension einer Aufgabe
$\boldsymbol{f}$	Äußere Kraft
$\boldsymbol{f}_n$	Äußere Kraft in die Normalenrichtung (Abschnitt 2.3)
$\boldsymbol{g}$	Äußere Kraft in den Randbedingungen
$\boldsymbol{g}_n$	Äußere Kraft in die Normalenrichtung (Abschnitt 2.3)
$\boldsymbol{g}_t$	Äußere Kraft in die Tangentialrichtung (Abschnitt 2.3)
h	Gitterparameter
m	Ordnung einer konformen Ansatzfunktion (Abschnitt 4.3)
m_{ij}^K	Abstand des Umkreiszentrums des Dreiecks K zum Mittelpunkt der Kante A_iA_j (Abschnitt 3.5)
m_{ij}	Länge der zur Kante A_iA_j senkrechten Linie zwischen zwei Umkreiszentren (Abschnitt 3.5)
$\boldsymbol{n}$, $\tilde{\boldsymbol{n}}$	Einheits-Normalvektor
n	Zeitschritt-Nummer
p, p_i	Druck, diskreter Druck
p_d	Potenz in einem Monom (Abschnitt 4.3)
r	Erste Komponente in zylindrischen Koordinaten (Abschnitt 2.2.2)
t	Zeit
$\triangle t$	Zeitschritt

$\boldsymbol{u}$	Geschwindigkeitsvektor
$\boldsymbol{u}_n$	Geschwindigkeitsvektor in die Normalenrichtung (Abschnitt 2.3)
$\boldsymbol{u}_t$	Geschwindigkeitsvektor in die Tangentialrichtung (Abschnitt 2.3)
u	Erste Komponente des Geschwindigkeitsvektors
$\bar{u}$	Dirichlet-Randbedingung oder Einström-Geschwindigkeit
u_i	Diskrete erste Geschwindigkeitskomponente im Punkt i des Gitters
v	Zweite Komponente des Geschwindigkeitsvektors
v_i	Diskrete zweite Geschwindigkeitskomponente im Punkt i des Gitters
w	Dritte Komponente des Geschwindigkeitsvektors
$\boldsymbol{x}$	Koordinatenvektor
x	Erste Komponente des Koordinatenvektors
y	Zweite Komponente des Koordinatenvektors
z	Dritte Komponente des Koordinatenvektors

Griechische Zeichen

Γ	Rand des Simulationsgebietes
Γ_D	Dirichlet-Rand
Γ_N	Neumann-Rand
Φ	Dissipationsfunktion (Kap. 2)
Φ	Abbildungsmatrix eines Gitter-Elements auf das Referenz-Element (Abschnitt 5.2.3)
Ω	Definitionsbereich
$\partial\Omega$	Rand des Definitionsbereiches
α, β, γ	Winkel eines allgemeinen Dreiecks (Abb. 3.7)
β	Wärmeausdehnungskoeffizient (Kap. 2)
η	Viskosität
η, η_K	Fehlerschätzung (Abschnitt 5.2.8)
θ	Parameter der Einschritt-θ-Methode
θ_{ij}	Gegenüber der Kante A_iA_j liegender Winkel eines Dreiecks (Abschnitt 3.5)
λ	Massenviskosität (Kap. 2)
ν	Kinematische Viskosität (Abschnitt 2.2)
$\boldsymbol{\xi}$	Referenz-Koordinatenvektor (Abschnitt 4.3)
ξ_i	Komponente des Referenz-Koordinatenvektors (Abschnitt 4.3)
ρ	Dichte
σ	Wärmekapazität (Kap. 2)
τ	viskoser Spannungstensor (Kap. 2)
φ	Zweite Komponente in zylindrischen Koordinaten (Abschnitt 2.2.2)
φ_i	Ansatzfunktion für die Geschwindigkeit (Abschnitt 3.4,4.3,5.2.3)
$\hat{\varphi}_i$	Ansatzfunktion für die Geschwindigkeit auf dem Referenz-Element (Abschnitt 4.3,5.2.3)
ψ_i	Ansatzfunktion für den Druck (Abschnitt 3.4,4.3,5.2.3)
$\hat{\psi}_i$	Ansatzfunktion für den Druck auf dem Referenz-Element (Abschnitt 4.3,5.2.3)

Abkürzungsverzeichnis

AF	Ansatzfunktion (Abschnitt 4.3,5.2.3)
DSL	engl. Domain Specific Language (Abschnitt 5.3.1)
GFEM	Galerkin-Finite-Elemente-Methode (Abschnitt 3.4)
GP	Generative Programmierung (Kap. 4)
GSS	engl. Generative Simulation Solver
MFLOPS	engl. millions of floating-point operations per second
FDM	Finite-Differenzen-Methode (Abschnitt 3.1.1)
FEM	Finite-Elemente-Methode (Abschnitt 3.4)
FVM	Finite-Volumen-Methode (Abschnitt 3.3)
LBB	Ladyzhenskaya-Babuška-Brezzi-Bedingung (Abschnitt 3.4)
NaSt	Navier-Stokes
OOP	Objektorientierte Programmierung (Abschnitt 1.2)
PDG	Partielle Differentialgleichung
STL	engl. Standard Template Library (Abschnitt 1.2)
UML	engl. Unified Modeling Language (Abschnitt 5.3.2,A.2.2)